



Quartus II Handbook, Volume 1

Design & Synthesis

ALTERA[®]

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
<http://www.altera.com>

Copyright © 2005 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. Design Flows

Revision History Section I-2

Chapter 1. Quartus II Design Flow for MAX+PLUS II Users

Introduction	1-1
Chapter Overview	1-1
Typical Design Flow	1-2
Device Support	1-3
Quartus II GUI Overview	1-4
Project Navigator	1-4
Node Finder	1-4
Tcl Console	1-4
Messages	1-4
Status	1-5
Setting up MAX+PLUS II Look & Feel in Quartus II	1-6
MAX+PLUS II Look & Feel	1-7
Compiler Tool	1-9
MAX+PLUS II Design Conversion	1-11
Converting an Existing MAX+PLUS II Design	1-11
Converting MAX+PLUS II Graphic Design Files	1-12
Importing MAX+PLUS II Assignments	1-13
Quartus II Design Flow	1-14
Creating a New Project	1-14
Design Entry	1-14
Making Assignments	1-18
Synthesis	1-21
Functional Simulation	1-21
Place & Route	1-23
Timing Analysis	1-24
Timing Closure Floorplan	1-26
Timing Simulation	1-27
Power Estimation	1-29

Programming	1–30
Conclusion	1–30
Quick Menu Reference	1–31
Quartus II Command Reference for MAX+PLUS II Users.....	1–33

Chapter 2. Quartus II Support of HardCopy Series Devices

Introduction	2–1
HardCopy II Device Support	2–1
HardCopy II Design Benefits	2–1
Quartus II Features for HardCopy II Planning	2–2
HardCopy II Prototyping Flow	2–2
HardCopy II Device Resource Guide	2–4
Migration Devices Dialog Box	2–6
Conclusion	2–9
HardCopy Stratix & HardCopy APEX Device Support	2–10
Features	2–11
HARDCOPY_FPGA_PROTOTYPE, HardCopy Stratix, & Stratix Devices	2–12
HardCopy Design Flow	2–14
The Design Flow Steps of the One Step Process	2–16
How to Design HardCopy Stratix Devices	2–16
Tcl Support for HardCopy Migration	2–21
Design Optimization & Performance Estimation	2–22
Design Optimization	2–22
Performance Estimation	2–22
Buffer Insertion	2–26
Placement Constraints	2–26
Location Constraints	2–27
LAB Assignments	2–27
LogicLock Assignments	2–28
Checking Designs for HardCopy Design Guidelines	2–29
Design Assistant Settings	2–30
Running Design Assistant	2–30
Reports and Summary	2–30
Generating the HardCopy Design Database	2–31
Static Timing Analysis (STA)	2–33
Power Estimation	2–33
HardCopy Stratix Power Estimator	2–33
Opening the HardCopy Stratix Power Estimator	2–34
HardCopy APEX Power Estimator	2–35
Tcl Support for HardCopy Stratix	2–35
Targeting Designs to HardCopy APEX Devices	2–36
Conclusion	2–36

Chapter 3. Engineering Change Management

Introduction	3–1
Impact of Last Minute Design Changes	3–1
Performance	3–1

Compile Time	3-2
Verification	3-2
Documentation	3-2
ECO Support	3-2
ECO Support at the HDL Level	3-3
ECO Support at the Netlist Level	3-5
Conclusion	3-6

Section II. Design Guidelines

Revision History	Section II-2
------------------------	--------------

Chapter 4. Design Recommendations for Altera Devices

Introduction	4-1
Synchronous FPGA Design Practices	4-1
Fundamentals of Synchronous Design	4-2
Hazards of Asynchronous Design	4-3
Design Guidelines	4-4
Combinational Logic Structures	4-4
Clocking Schemes	4-8
Hierarchical Design Partitioning	4-14
Targeting Clock & Register-Control Architectural Features	4-15
Clock Network Resources	4-15
Reset Resources	4-17
Register Control Signals	4-17
Conclusion	4-17

Chapter 5. Recommended HDL Coding Styles

Introduction	5-1
Instantiating & Inferring Altera Megafunctions	5-1
Instantiating Altera Megafunctions in HDL Code	5-2
Inferring Megafunctions from HDL Code	5-4
Inferring Counters from HDL Code	5-5
Inferring Adder/Subtractors from HDL Code	5-7
Inferring Multipliers from HDL Code	5-8
Inferring Multiply-Accumulators & Multiply-Adders from HDL Code	5-11
Inferring RAM from HDL Code	5-14
Inferring ROM from HDL Code	5-19
Inferring Shift Registers from HDL Code	5-22
Device-Specific Coding Recommendations	5-25
Secondary Control Signals in Registers or Flipflops	5-25
Tri-State Signals	5-28
Adder Trees	5-28
General Coding Recommendations	5-31
Latches	5-31
State Machines	5-32

Multiplexers	5–38
Conclusion	5–47

Section III. Synthesis

Revision History	Section III–2
------------------------	---------------

Chapter 6. Quartus II Integrated Synthesis

Introduction	6–1
Language Support	6–1
Verilog HDL	6–1
VHDL	6–2
AHDL	6–5
Schematic Design Entry	6–6
Design Flow	6–6
Incremental Synthesis	6–8
Partitions for Incremental Synthesis	6–8
Preparing a Design for Incremental Synthesis	6–10
Synthesizing a Design Using Incremental Synthesis	6–10
Forcing Complete Re-synthesis	6–12
Considerations & Restrictions When Using Incremental Synthesis	6–13
Types of Synthesis Options	6–16
Synthesis Directives	6–17
Synthesis Attributes	6–17
Quartus II Logic Options	6–19
Quartus II Synthesis Options	6–20
Translate Off & On	6–20
Read Comments as HDL	6–21
Full Case	6–22
Parallel Case	6–23
Keep Combinational Node/Implement as Output of Logic Cell	6–24
Preserve Registers	6–25
Maximum Fan-Out	6–26
Optimization Technique	6–28
State Machine Processing	6–28
Preserve Hierarchical Boundary	6–30
Restructure Multiplexers	6–30
Power-Up Level	6–32
Power-Up Don't Care	6–33
Remove Duplicate Logic	6–33
Remove Duplicate Registers	6–34
Remove Redundant Logic Cells	6–34
Megafunction Inference Control	6–35
RAM Style	6–37
Multiplier Style	6–38
Setting Other Quartus II Options in Your HDL Source Code	6–40
Use I/O Flip-Flops or Registers	6–40

Altera Attribute	6-42
Chip Pin	6-44
Node-Naming Conventions in Quartus II Integrated Synthesis	6-46
Hierarchical Node-Naming Conventions	6-46
Node-Naming Conventions for Registers (DFF or D Flip-Flop Atoms)	6-47
Registers That Can Change During Synthesis	6-48
Node-Naming Conventions for Combinational Logic Cells	6-49
Scripting Support.....	6-51
Quartus II Synthesis Options	6-51
Assigning a Pin	6-53
Preparing a Design for Incremental Synthesis	6-53
Conclusion	6-54

Chapter 7. Synplify Synplify & Synplify Pro Support

Introduction	7-1
Design Flow	7-1
Synplify Optimization Strategies	7-5
Implementations in Synplify Pro	7-6
Timing-Driven Synthesis Settings	7-6
Finite State Machine (FSM) Compiler	7-9
General Optimization Attributes & Options	7-10
Altera Specific Attributes	7-11
Exporting Designs to the Quartus II Software Using NativeLink Integration	7-13
Running the Quartus II Software from within the Synplify Software	7-14
Using the Quartus II Software to Launch the Synplify Software	7-14
Cross-Probing with the Quartus II Software	7-15
Enabling Cross-Probing	7-15
Cross-Probing from the Quartus II Software	7-16
Cross-Probing from the Synplify Software	7-16
Guidelines for Altera Megafunctions & Architecture-Specific Features	7-17
Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager	7-18
Inferring Altera Megafunctions from HDL Code	7-22
Block-Based Design with the Quartus II LogicLock Methodology	7-27
Hierarchy & Design Considerations with Multiple VQM Files	7-28
Creating a Design with Multiple VQM Files	7-28
Creating a Design with Multiple VQM Files Using Synplify Pro Multipoint Synthesis	7-29
Generating a Design with Multiple VQM Files Using Black Boxes	7-35
Conclusion	7-40

Chapter 8. Mentor Graphics LeonardoSpectrum Support

Introduction	8-1
Design Flow	8-1
Optimization Strategies	8-4
Timing-Driven Synthesis	8-4
Other Constraints	8-5
Timing Analysis with the Leonardo-Spectrum Software	8-7
Exporting Designs Using NativeLink Integration	8-8

Generating Netlist Files	8-8
Including Design Files for Black-Boxed Modules	8-8
Passing Constraints with Scripts	8-8
Integration with the Quartus II Software	8-9
Guidelines for Altera Megafunctions & LPM Functions	8-9
Inferring Multipliers & DSP Functions	8-11
Controlling DSP Block Inference	8-12
Block-Based Design with the Quartus II LogicLock Methodology	8-18
Hierarchy & Design Considerations	8-18
Creating a Design with Multiple EDIF Files	8-19
Generating Multiple EDIF Files Using Black Boxes	8-24
Incremental Synthesis Flow	8-29
Conclusion	8-32

Chapter 9. Mentor Graphics Precision RTL Synthesis Support

Introduction	9-1
Design Flow	9-1
Creating a Project & Compiling the Design	9-5
Creating a Project	9-5
Compiling the Design	9-6
Setting Constraints	9-6
Setting Timing Constraints	9-7
Setting Mapping Constraints	9-7
Assigning Pin Numbers & I/O Settings	9-8
Assigning I/O Registers	9-9
Disabling I/O Pad Insertion	9-9
Controlling Fan-Out on Data Nets	9-10
Synthesizing the Design & Evaluating the Results	9-11
Obtaining Accurate Logic Utilization & Timing Analysis Reports	9-11
Exporting Designs to the Quartus II Software Using NativeLink Integration	9-12
Megafunctions & Architecture-Specific Features	9-14
Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager	9-15
Inferring Altera Megafunctions from HDL Code	9-17
Block-Based Design with the Quartus II LogicLock Methodology	9-23
Hierarchy & Design Considerations	9-23
Creating a Design with Separate Blocks for the LogicLock Methodology	9-24
Creating a Design with Separate Blocks Using the LogicLock Attribute in a Single Precision Project	9-24
Generating a Design with Multiple EDIF Files Using Black Boxes	9-26
Conclusion	9-30

Chapter 10. Synopsys FPGA Compiler II BLIS & Quartus II LogicLock Design Flow

Introduction	10-1
Design Hierarchy	10-1
Block-Level Incremental Synthesis	10-2
FPGA Compiler II Design Block	10-2
FPGA Compiler II & Quartus II Synthesis	10-3
Block Root	10-3

How the BLIS Feature Works with the LogicLock Feature	10-4
Hierarchy Considerations	10-5
Time Stamp Synthesis	10-6
Creating & Maintaining a Design	10-6
Opening the Modules Constraint Table & Labeling Block Roots	10-7
Exporting Block-Level Netlist Files	10-7
Changing Source Within a Block	10-8
Removing a Block Root	10-9
Using BLIS Shell Commands	10-9
Conclusion	10-10

Chapter 11. Synopsys Design Compiler FPGA Support

Introduction.....	11-1
Design Flow Using the DC FPGA Software & the Quartus II Software	11-2
Setup of the DC FPGA Software Environment for Altera Device Families	11-3
Megafunctions & Architecture-Specific Features	11-4
Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager	11-5
Reading Megafunction Wizard-generated Variation Wrapper Files	11-5
Using Megafunction Wizard-generated Variation Wrapper Files in a Black-Box Methodology	11-6
Inferring Altera Megafunctions from HDL Code	11-7
Reading Design Files into the DC FPGA Software	11-9
Selecting a Target Device	11-11
Timing & Synthesis Constraints	11-13
Compilation & Synthesis	11-14
Reporting Design Information	11-15
Saving Synthesis Results	11-16
Exporting Designs to the Quartus II Software	11-17
Place & Route with the Quartus II Software	11-19
Conclusion	11-19

Chapter 12. Analyzing Designs with the Quartus II RTL Viewer & Technology Map Viewer

Introduction	12-1
RTL Viewer Overview	12-1
Technology Map Viewer Overview	12-2
Quartus II Design Flow with the RTL & Technology Map Viewers	12-3
Introduction to the User Interface	12-4
Schematic View	12-5
Hierarchy List	12-13
Navigating the Schematic View	12-14
Zooming & Magnification	12-14
Partitioning the Schematic into Pages	12-15
Traversing the Design Hierarchy	12-18
Moving Back & Forward Through Schematic Pages	12-19
Go to Net Driver	12-19
Filtering in the Schematic View	12-19
Examples of Filtered Netlists	12-21

Filtering Across Hierarchies	12–22
Expanding a Filtered Netlist	12–24
Reducing a Filtered Netlist	12–24
Probing to Source Design File & Other Quartus II Windows	12–25
Probing to the Viewers from Other Quartus II Windows	12–25
Viewing a Timing Path in the Technology Map Viewer	12–26
Other Features in the Schematic Viewer	12–27
Tooltips	12–27
Displaying Net Names	12–30
Full Screen View	12–30
Find Command	12–30
Exporting Schematic as JPEG or BMP Image & Copying to Clipboard	12–31
Printing	12–32
Using the RTL & Technology Map Viewers to Analyze Design Problems	12–32
Conclusion	12–33

Index



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. Quartus II Design Flow for MAX+PLUS II Users

Revised: December 2004

Part number: *qii51002-2.1*

Chapter 2. Quartus II Support of HardCopy Series Devices

Revised: January 2005

Part number: *qii51004-2.1*

Chapter 3. Engineering Change Management

Revised: June 2004

Part number: *qii51005-2.0*

Chapter 4. Design Recommendations for Altera Devices

Revised: December 2004

Part number: *qii51006-2.1*

Chapter 5. Recommended HDL Coding Styles

Revised: December 2004

Part number: *qii51007-2.1*

Chapter 6. Quartus II Integrated Synthesis

Revised: December 2004

Part number: *qii51008-3.0*

Chapter 7. Synplicity Synplify & Synplify Pro Support

Revised: December 2004

Part number: *qii51009-2.1*

Chapter 8. Mentor Graphics LeonardoSpectrum Support

Revised: December 2004

Part number: *qii51010-2.1*

Chapter 9. Mentor Graphics Precision RTL Synthesis Support

Revised: December 2004

Part number: *qii51011-2.1*

Chapter 10. Synopsys FPGA Compiler II BLIS & Quartus II LogicLock Design Flow

Revised: *June 2004*

Part number: *qii51012-1.0*

Chapter 11. Synopsys Design Compiler FPGA Support

Revised: *December 2004*

Part number: *qii51014-1.1*

Chapter 12. Analyzing Designs with the Quartus II RTL Viewer & Technology Map Viewer

Revised: *December 2004*

Part number: *qii51013-2.1*



About this Handbook

This handbook provides comprehensive information about the Altera® Quartus®II design software, version 4.2.

How to Contact Altera

For the most up-to-date information about Altera products, go to the Altera world-wide web site at www.altera.com. For technical support on this product, go to www.altera.com/mysupport. For additional information about Altera products, consult the sources shown below.

Information Type	USA & Canada	All Other Locations
Technical support	www.altera.com/mysupport/	altera.com/mysupport/
	(800) 800-EPLD (3753) (7:00 a.m. to 5:00 p.m. Pacific Time)	(408) 544-7000 (1) (7:00 a.m. to 5:00 p.m. Pacific Time)
Product literature	www.altera.com	www.altera.com
Altera literature services	(1)	(1)
Non-technical customer service	(800) 767-3753	(408) 544-7000 (7:30 a.m. to 5:30 p.m. Pacific Time)
FTP site	ftp.altera.com	ftp.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 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: f_{MAX} , \qdesigns directory, d: drive, chiptrip.gdf file.
<i>Italic Type with Initial Capital Letters</i>	Document titles are shown in italic type with initial capital letters. Example: <i>AN 75: High-Speed Board Design</i> .

Visual Cue	Meaning
<i>Italic type</i>	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	The caution indicates required information that needs special consideration and understanding and should be read prior to starting or continuing with the procedure or process.
	The warning indicates information that should be read prior to starting or continuing the procedure or processes
↔	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 4.2 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 4.2 software options that are available for each of these flows.

This section includes the following chapters:

- Chapter 1, Quartus II Design Flow for MAX+PLUS II Users
- Chapter 2, Quartus II Support of HardCopy Series Devices
- Chapter 3, Engineering Change Management

Revision History

Chapter 1, *Hierarchical Black-Based & Team-Based Design Flows* was removed from this handbook. The table below shows the revision history for Chapters 1 to 3.

Chapter(s)	Date / Version	Changes Made (Part 1 of 3)
1	Dec. 2004 v2.1	<p>Updated for Quartus II software version 4.2.</p> <ul style="list-style-type: none"> ● Chapter 1 was formerly Chapter 2 in version 4.1. ● General formatting and editing updates. ● Device support for Quartus II software now includes FLEX 6000. See Table 1–1. ● Updated Figures 1–3, 1–7, 1–9, and 1–21. ● A new paragraph was added to the description of the “Assignment Editor”. ● “Timing Assignments” was updated to describe time groups on page 1–20. ● “Synthesis” was updated to describe “advanced integrated synthesis.” ● APEX II is no longer supported in the balanced optimization technique. Support was added for MAX II. See page 1–21. ● Minor updates were made to the description of “Place & Route” ● Tcl commands are no longer supported for the “Quartus II Simulator Tool”. ● The description of supported files was updated for “EDA Timing Simulation”. Table 1–3 was added to illustrate the file support. ● Description of the Excel-based power calculator was removed and was replaced by the PowerPlay Early Power Estimation spreadsheet. See “Power Estimation”. ● In “Programming” there is new support for an erase capability for CPLDs. The erase capability is shown in Figure 1–21.
	June 2004 v2.0	<ul style="list-style-type: none"> ● Updates to tables, figures. ● New functionality for Quartus II software 4.1.
	Feb. 2004 v1.0	Initial release.
2	Dec. 2004 v2.1	<ul style="list-style-type: none"> ● Chapter 2 was formerly Chapter 3. ● Updates to tables, figures. ● New functionality for Quartus II software 4.2
	June 2004 v2.0	<ul style="list-style-type: none"> ● Updates to tables, figures. ● New functionality for Quartus II software 4.1.
	Feb. 2004 v1.0	Initial release.

Chapter(s)	Date / Version	Changes Made (Part 2 of 3)
	Dec. 2004 v2.1	<p>Updated for Quartus II software version 4.2:</p> <ul style="list-style-type: none"> ● Chapter 3 was formerly Chapter 4 in version 4.1. ● General formatting and editing updates. ● Throughout the chapter, the text more clearly describes support for each device family. ● The description of HardCopy structured support for performance improvements was updated in the first bullet on page 2–10. ● In the Quartus II software version 4.2, the Quartus II Archive File file now automatically receives buffer insertion information, see “Improved Timing Estimation”. ● The HardCopy Stratix Power Calculator is now called the HardCopy Stratix Power Estimator. ● The HardCopy APEX 20K Power Calculator is now called the HardCopy APEX Power Estimator. ● Note 1 for Table 2–3 was updated to describe combinational and registered logic. ● Figure 2–6 was updated to clarify support for APEX devices. ● The description of “How to Design HardCopy Stratix Devices” was updated. In the procedure to target a design to a HardCopy Stratix device, Step 3 was updated. ● The description of “HardCopy Timing Optimization Wizard” was updated. ● “HardCopy Floorplans & Timing Modules” was renamed to “Design Optimization”. ● The description of “Performance Estimation” was updated. In the procedure to perform Timing Analysis for a hardCopy device, Step 3 was updated. ● Table 2–4 was added. It lists the directory structure generated by the HardCopy Timing Wizard. ● A new section on “Buffer Insertion” was added. ● Figure 2–11 was updated to clarify that it is for HardCopy Stratix devices. ● “Location Constraints” was updated. ● “Targeting Designs to HardCopy APEX 20KC and HardCopy APEX 20KE Devices” was removed. ● A new section “Altera Recommended HDL Coding Guidelines” was added.

Chapter(s)	Date / Version	Changes Made (Part 3 of 3)
		<ul style="list-style-type: none">● A new section “Altera Recommended HDL Coding Guidelines” 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 on “Targeting Designs to HardCopy APEX Devices” was added.
	June 2004 v2.0	<ul style="list-style-type: none">● Updates to tables, figures.● New functionality for Quartus II software 4.1.
	Feb. 2004 v1.0	Initial release.
3	Jan. 2005 v2.1	<ul style="list-style-type: none">● Added HardCopy II Device Material
	Dec. 2004 v2.0	<ul style="list-style-type: none">● Chapter 4 was formerly Chapter 5 in version 4.1.● Updates to tables, figures.● New functionality for Quartus II software 4.2
	June 2004 v2.0	No change to document.
	Feb. 2004 v1.0	Initial release.

qii51002-2.1

Introduction

The feature-rich Quartus® II software enables you to shorten your design cycles and achieve a reduced time-to-market. With Stratix® II, Stratix GX, Stratix, and MAX® II family support, the Quartus II software is the most widely accepted Altera® design software tool today.

This chapter describes a simple process for converting MAX+PLUS® II designs to Quartus II projects, as well as similarities and differences between the MAX+PLUS II design flow and the Quartus II design flow. This includes supported device families, 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 make an easy transition to the Quartus II software design environment. These include an option in the Quartus II software that causes the GUI to display menus, tool bars, and utility windows as they appear in the MAX+PLUS II software without sacrificing functionality.

Chapter Overview

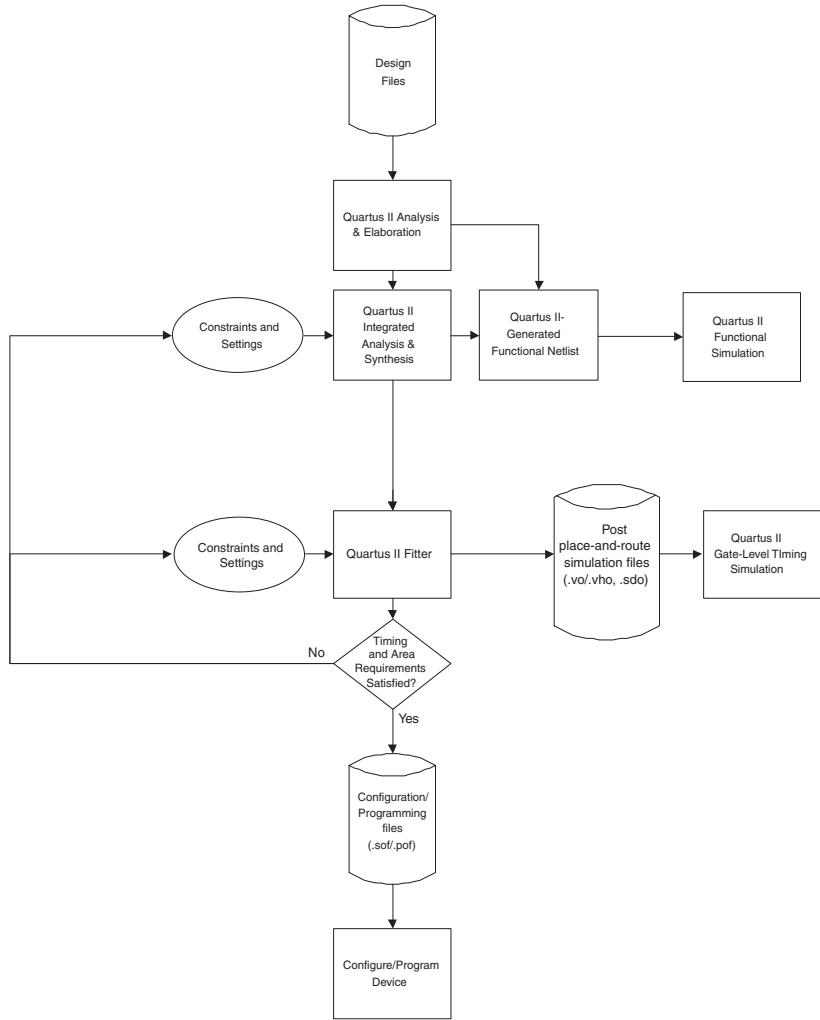
This chapter covers the following topics:

- “Chapter Overview”
- “Typical Design Flow”
- “Device Support”
- “Quartus II GUI Overview”
- “Setting up MAX+PLUS II Look & Feel in Quartus II”
- “Compiler Tool”
- “Quartus II Design Flow”
- “Conclusion”
- “Quick Menu Reference”

Typical Design Flow

Figure 1–1 shows a typical design flow with the Quartus II software.

Figure 1–1. 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 1–1](#).

Table 1–1. Device Support Comparison

Device Supported	Quartus II	MAX+PLUS II
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™ 20K/ APEX II	✓	
Stratix	✓	
Stratix GX	✓	
Stratix II	✓	
Cyclone™	✓	
Cyclone II	✓	

Notes to Table 1–1:

- (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, see 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, warning, and error messages. It also allows you 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.

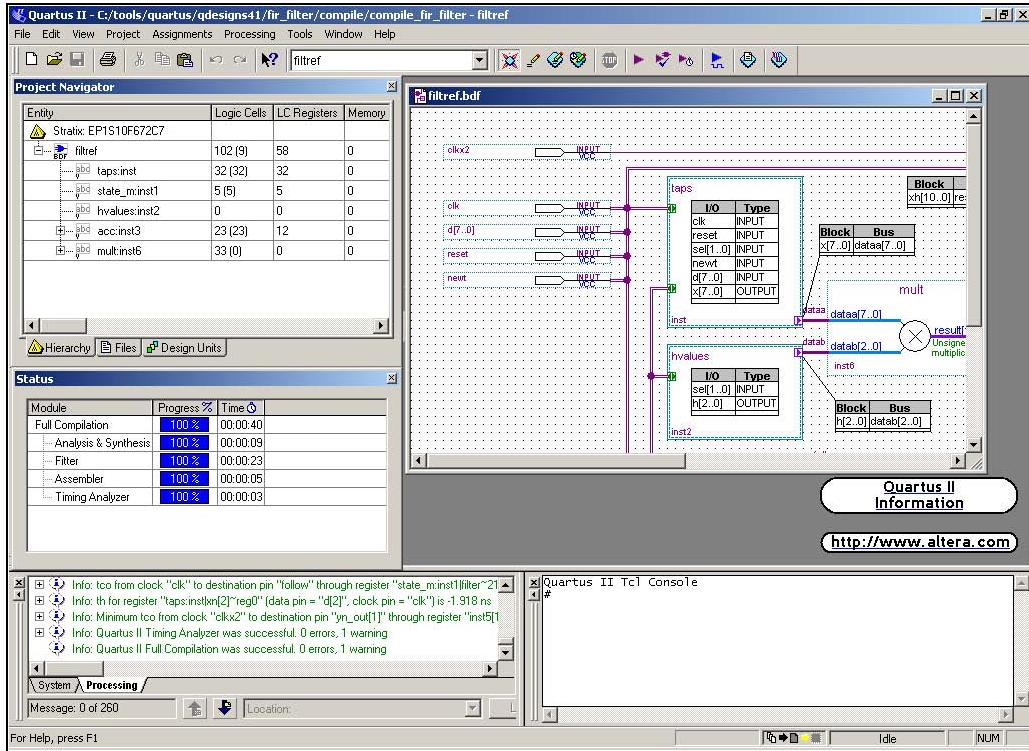
Change Manager

The Change Manager provides detailed tracking information on all design changes made with the Chip Editor.

For more information on the Engineering Change Manager and the Chip Editor, see the *Design Analysis and Engineering Change Management with Chip Editor* chapter in Volume 3 of the *Quartus II Handbook*.

The Quartus II software is shown in [Figure 1–2](#).

Figure 1–2. Example of the Quartus II Look & Feel

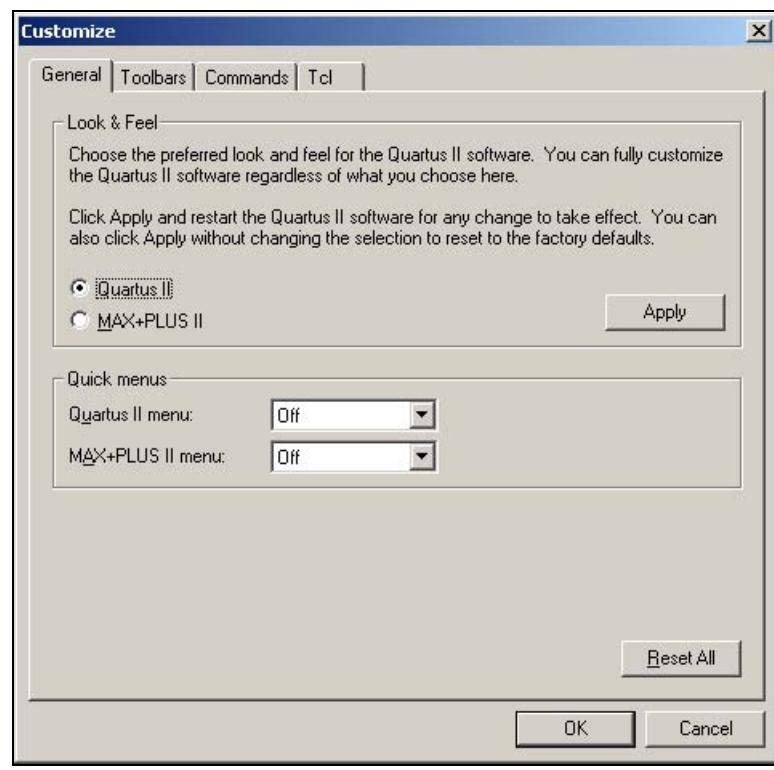


Setting up MAX+PLUS II Look & Feel in Quartus II

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 (Tools menu). Any changes to the look and feel do not take effect 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 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 can also change the preferred positions of the two quick menus. These options are available in the **Quick menus** box of the **General** tab of the **Customize** dialog box (Tools menu). To restore the factory default settings, click **Apply** without changing any of the selections (see Figure 1–3).

Figure 1–3. Customize Dialog Box—General Tab



MAX+PLUS II Look & Feel

The MAX+PLUS II look and feel of the Quartus II software closely resembles the MAX+PLUS II software. Figures 1–4 and 1–5 compare the appearance of the MAX+PLUS II look and feel.

Figure 1–4. MAX+PLUS II Software GUI

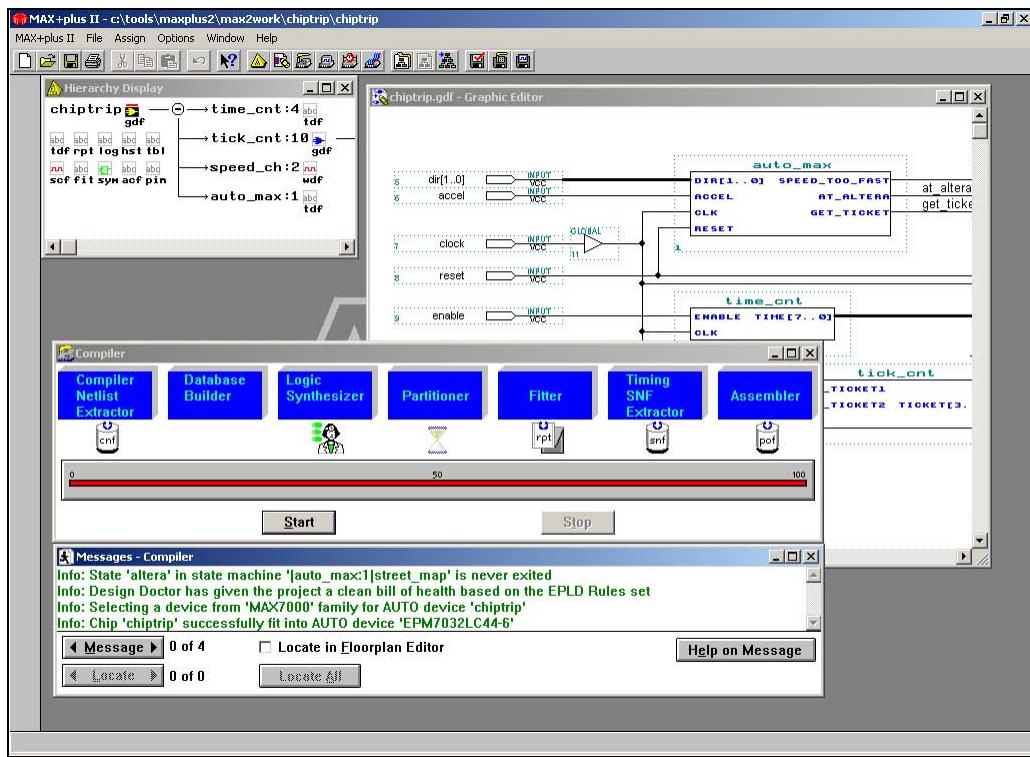
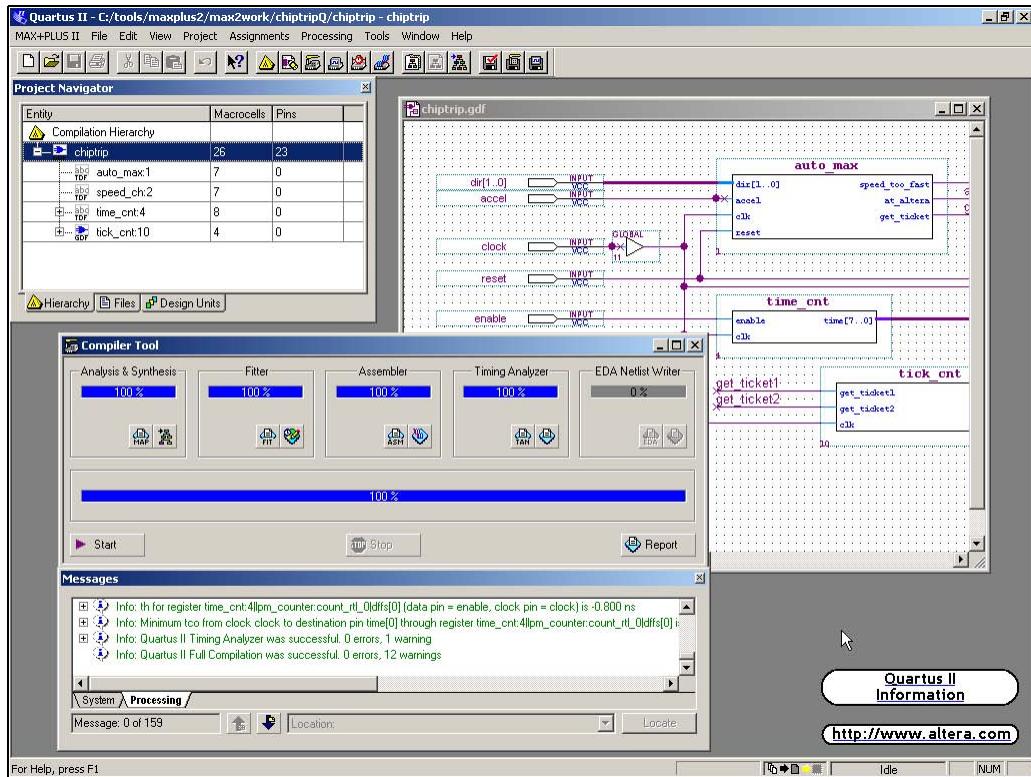


Figure 1–5. Quartus II Software with MAX+PLUS II Look & Feel



The standard MAX+PLUS II tool bar is also available in the Quartus II software with the MAX+PLUS II look and feel (see [Figure 1–6](#)).

Figure 1–6. Standard MAX+PLUS II Tool Bar



Compiler Tool

The Compiler Tool provides an intuitive MAX+PLUS II-style interface. You can edit the settings and view result files for the following modules:

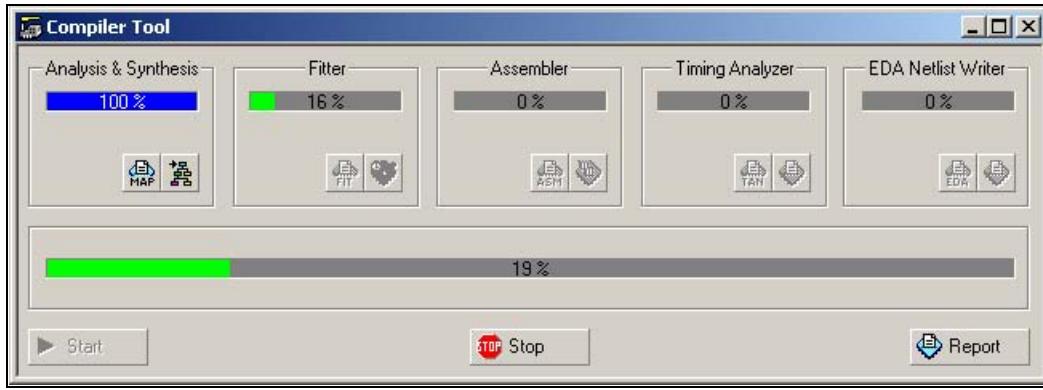
- Analysis and Synthesis
- Fitter
- Assembler
- Timing Analyzer
- EDA Netlist Writer

To start a compilation using the Compiler Tool, choose **Compiler Tool** from either the MAX+PLUS II menu or the Tools menu and click **Start** in the Compiler Tool (see [Figure 1–7](#)).



For information about using the Quartus II software modules at the command line, see the *Command-Line Scripting* chapter in Volume 2 of the *Quartus II Handbook*.

Figure 1–7. Running a Full Compilation with the Compiler Tool



The Analysis and Synthesis module analyzes your design to build the design database, optimizes it for the targeted architecture, and performs technology mapping on the design logic. These are the functions

performed by the Compiler Netlist Extractor, Database Builder, and Logic Synthesizer in the MAX+PLUS II software. There is no module in the Quartus II software similar to the MAX+PLUS II Partitioner module.

The 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.

The 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)

The Assembler module is similar to the Assembler stage of the MAX+PLUS II software.

The EDA Netlist Writer module generates a netlist for simulation with an EDA simulation tool. The EDA Netlist Writer module is comparable to the VHDL+Verilog Netlist Writer stage of the MAX+PLUS II software.

You can significantly reduce subsequent compilation times in the Quartus II software if you turn on **Smart Compilation** in the **Compilation Process** page in the **Settings** dialog box (Assignments menu). The Smart Compilation feature skips any compilation stages that are not required but may use more disk space. This option is similar to the MAX+PLUS II **Smart Recompile** command.

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 so on) are associated with a project file. For more information about creating a new project, see “[Creating a New Project](#)” on page 1–14.

Converting an Existing MAX+PLUS II Design

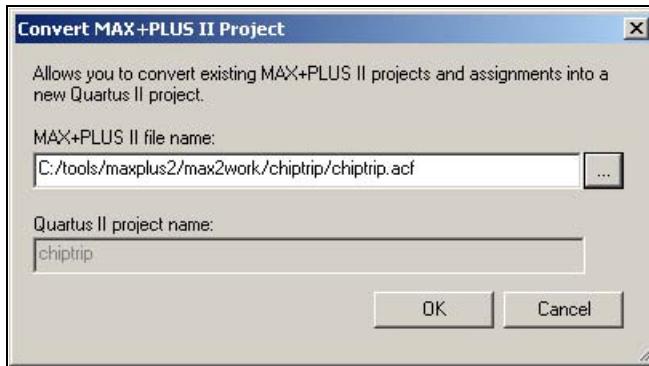
You can easily convert an existing MAX+PLUS II design for use with the Quartus II software with the **Convert MAX+PLUS II Project** (File menu) commands in the Quartus II software or the **Open Project** command (File menu).

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 (see [Figure 1–8](#)) and click Open. The 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 QSF, equivalent to the ACF in the MAX+PLUS II software.

You can also open and convert a MAX+PLUS II design with the **Open Project** command. In the **Open Project** dialog box, browse to the ACF or the top-level design file. Click **Open** to bring up 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 1–8. Convert MAX+PLUS II Project Dialog Box

The conversion process performs the following actions:

- Converts the ACF into a QSF (equivalent to importing all MAX+PLUS II assignments)
- Creates a **.qpf** file
- Displays all errors and warnings in the messages 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 GDF into a BDF using one of the following methods:

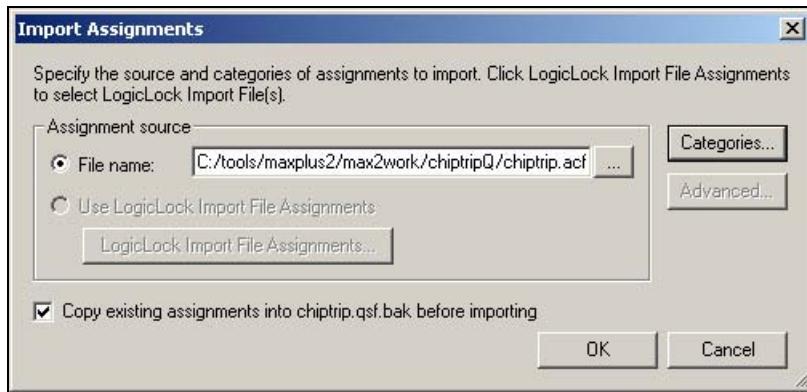
- Open the GDF and choose **Save As** (File menu). In the **Save As** dialog box, choose **Block Diagram/Schematic File (*.bdf)** from the **Save as type** list.
- Run the **quartus_g2b.exe** command line executable located in the **/Quartus II installation/bin** directory. For example, to convert the **chiptrip.gdf** file to a BDF, 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, choose **Import Assignments** (Assignments menu), and browse to the ACF (see [Figure 1–9](#)). You can also import QSF and Entity Setting Files (.esf).

Figure 1–9. Import Assignments Dialog Box



The Quartus II software accepts most MAX+PLUS II assignments. However, it is possible for an assignment to be imported incorrectly due to node name formats.

The format of node names is different in the Quartus II and MAX+PLUS II software. Make sure that the naming schemes map properly and do not interfere with design logic. [Table 1–2](#) compares the differences between the naming conventions used by the Quartus II software and the MAX+PLUS II software.

Table 1–2. Quartus II & MAX+PLUS II Node & Pin Naming Schemes

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 that contain node names that use numbers, such as signal10 or signal11, the Quartus II software inserts square brackets 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.

If your MAX+PLUS II design contains node names that end in a number and are not part of a bus, you must edit the QSF to remove the square brackets from the node names after importing them.

The Quartus II software and the MAX+PLUS II software synthesize nodes differently. 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.

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.

Creating a New Project

The Quartus II software provides a wizard to help you create new projects. Choose **New Project Wizard** (File menu) to start the New Project Wizard. The New Project Wizard generates the QPF and QSF for your project.

Design Entry

The Quartus II software supports the following design entry methods:

- AHDL (.tdf)
- Block Diagram File (.bdf)
- EDIF Netlist File (.edf)
- Verilog Quartus Mapping Netlist File (.vqm)
- VHDL (.vhd)
- Verilog HDL (.v)

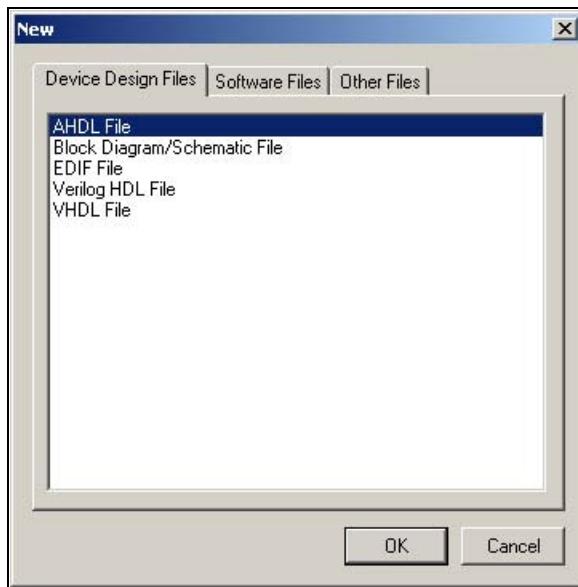
The Quartus II software has an advanced integrated synthesis engine that fully supports the Verilog HDL and VHDL languages and provides options to control the synthesis process.



For more information, see the *Quartus II Integrated Synthesis* chapter in Volume 1 of the *Quartus II Handbook*.

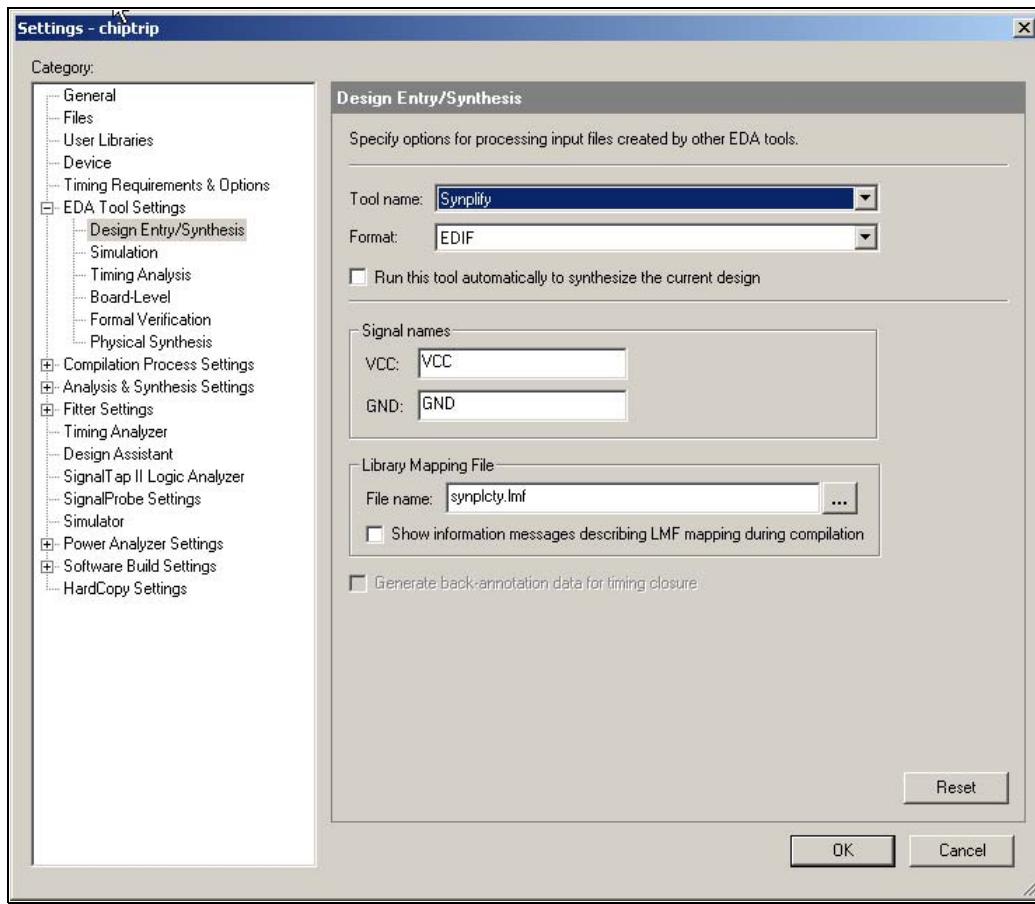
To create a new design file, select a design entry type in the **Device Design Files** tab of the **New** dialog box (File menu) and click **OK** (see [Figure 1–10](#)).

Figure 1–10. New Dialog Box



You can create other files from the **Software Files** tab and **Other Files** tab of the **New** dialog box (File menu). For example, the Vector Waveform File (.vWF) is located in the **Other Files** tab.

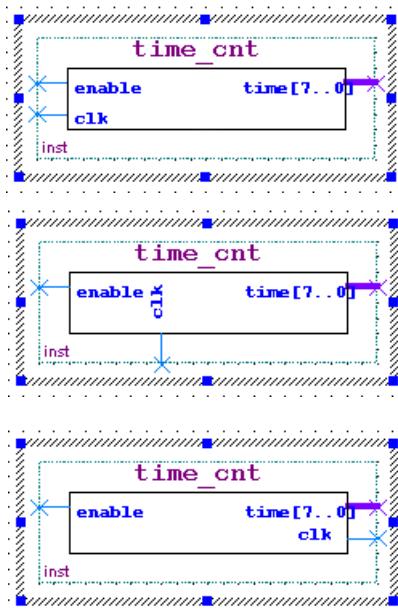
To analyze a netlist file created by an EDA tool, select the synthesis tool used to generate it in the **Tool** name list of the **Design Entry & Synthesis** page under **EDA Tool Settings** in the **Settings** dialog box (Assignments menu). See [Figure 1–11](#).

Figure 1–11. Settings Dialog Box

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.

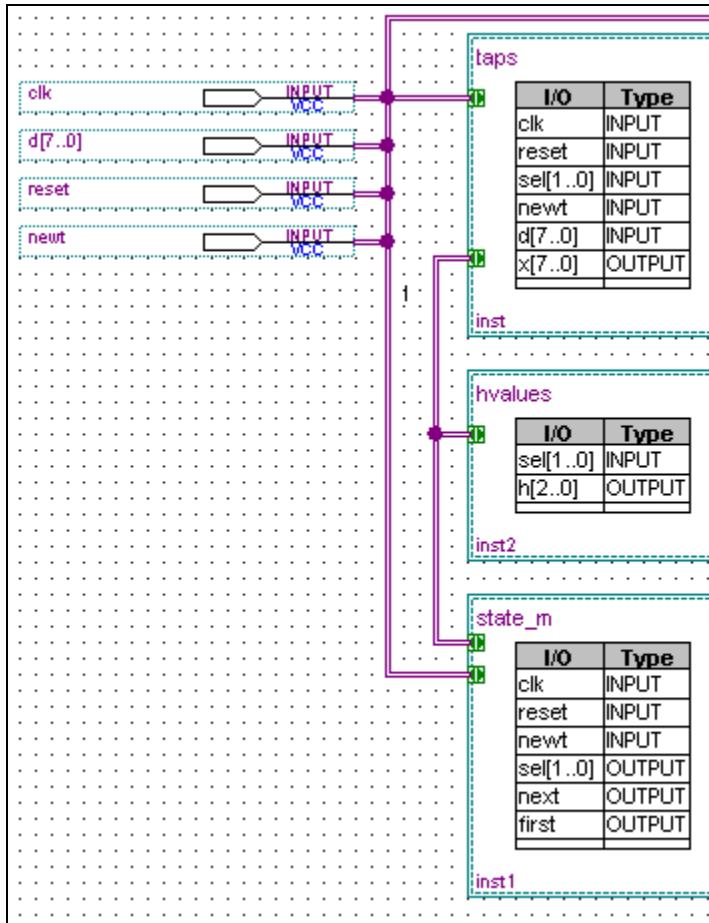
The Symbol Editor allows you to change the positions of the ports in a symbol (see the three images in [Figure 1–12](#)). You can reduce wire congestion around a symbol by changing the positions of the ports.

Figure 1–12. Various Port Position for a Symbol



To make changes to a symbol in a BDF, right-click on a symbol in the Block Editor and select **Properties** (right button pop-up menu) to bring up 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 1–13](#)). You can determine the connections between various blocks in the **Conduit Properties** dialog box by right clicking a conduit and choosing **Properties** (right button pop-up menu).

Figure 1–13. Blocks & Pins Connected with Conduits

Making Assignments

The Quartus II software stores all project and design assignments in a QSF. The QSF is a collection of assignments stored as Tcl commands and organized by compilation stage and assignment type. The QSF stores all assignments, regardless of how they are made: from the Floorplan Editor, the Assignment Editor, with Tcl, or any other method.

Assignment Editor

The Assignment Editor is an intuitive spreadsheet interface designed to allow you to easily make, change, and manage a large number of assignments. With the Assignment Editor, you can list all available pin numbers and design pin names for efficiently creating pin assignments. You can also 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, perform the following steps:

1. Choose **Assignment Editor** (Assignments menu) to open the Assignment Editor.
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. Choose **Save** (File menu).

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, see 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. Choose **Timing Wizard** (Assignments menu) 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 can also set timing settings manually on the **Timing Requirements & Options** page of the **Settings** dialog box (Assignments menu).

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 or two registers or between 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 1–14](#) shows a 4 ns t_{SU} assignment to a bus of registers made in the Assignment Editor.

Figure 1–14. Single t_{SU} Timing Assignment Applied to All Nodes of a Bus

	From	To	Assignment Name	Value
1	◊ *	◊ d[?]	t_{SU} Requirement	4ns
2	<<new>>	<<new>>	<<new>>	<<new>>



For more information, see the *Timing Analysis* chapter in Volume 3 of the *Quartus II Handbook*.

Synthesis

The Quartus II software includes advanced integrated synthesis that fully supports the Verilog and VHDL hardware description languages (HDLs), 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, stand-alone solution for today's designs.

You can specify several synthesis options in the **Analysis & Synthesis Settings** page of the **Settings** dialog box. Similar to MAX+PLUS II synthesis options, you can select **Speed**, **Area**, or **Balanced** for the optimization technique.

 Only the APEX 20K, MAX II, Cyclone, Cyclone II, Stratix II, and Stratix device families support the balanced optimization technique.

To achieve higher 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 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 (Assignments menu).

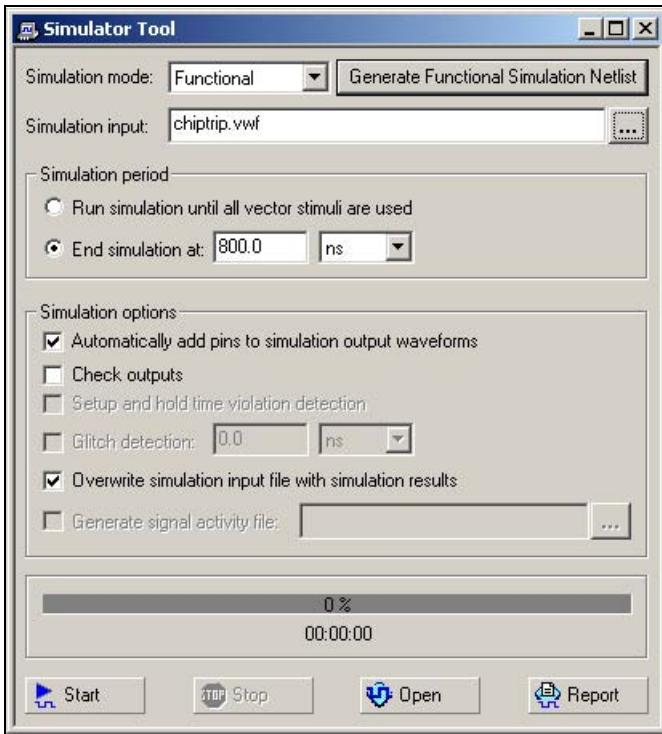
 For more information, see the *Quartus II Integrated Synthesis* chapter in Volume 1 of the *Quartus II Handbook*.

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, choose **Simulator** (MAX+PLUS II menu) or **Simulator Tool** (Tools menu). Before you perform a functional simulation, a functional simulation netlist is required. Click **Generate Functional Simulation Netlist** in the **Simulator Tool** window (see [Figure 1-15](#)) or choose **Generate Functional Simulation Netlist** (Processing menu).

 Generating a functional simulation netlist creates a separate database to significantly improve the performance of the simulation.

Figure 1–15. Simulator Tool Dialog Box

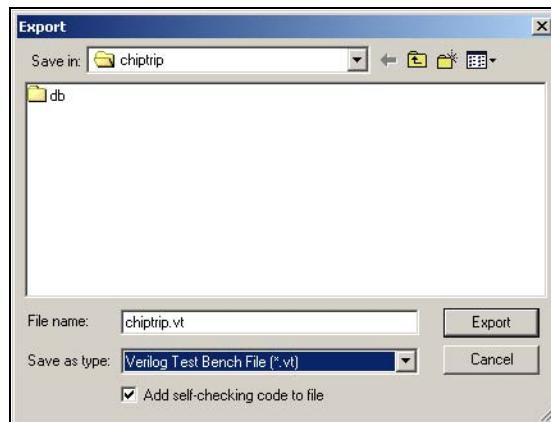
You can view and modify the simulator options on the **Simulator** page of the **Settings** dialog box or in the **Simulator Tool** window. You can set the simulation period and turn **Check outputs** on or off. You can choose to display the simulation outputs in the simulation report or in the Vector Waveform File (.vWF). 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 Install/eda/sim_lib` directory.

The Quartus II schematic design files (BDF) are not compatible with EDA simulation tools. To perform a register transfer level (RTL) functional simulation of a BDF using an EDA tool, convert your schematic designs to a VHDL or Verilog HDL design file. Open the schematic design file and choose **Create/Update > Create HDL Design File for Current File** (File menu) to create an HDL design file that corresponds to your BDF.

You can export a VWF or SCF simulation file as a Verilog HDL or VHDL testbench file for simulation with an EDA tool. Open your VWF or SCF file and choose **Export** (File menu) (see [Figure 1–16](#)). 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 testbench.

Figure 1–16. Export Dialog Box



Place & Route

The Quartus II PowerFit fitter performs place-and-route to fit your design into the targeted device.

You can control the Fitter behavior with options on the **Fitter Settings** page of the **Settings** dialog box (Assignments menu).

High-density device families supported in the Quartus II software, such as Stratix devices, sometimes require significant fitter effort to process. 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 two options: **Optimize Timing** and **Optimize I/O cell register placement for timing options**. By default,

both options are turned on; however, if compilation time is more important than achieving specific timing results, you can turn those options off.

You can control the amount of effort the Fitter expends to fit your design by selecting **Standard Fit** or **Fast Fit** in the **Fitter Effort** box of the **Fitter Settings** page in the **Settings** dialog box (Assignments menu). Select **Standard Fit** to have the Fitter use the highest effort, preserving the performance from previous compilations. Select **Fast Fit** for up to 50% faster compilation times, though this may cause a reduction in 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 Stratix II, Stratix GX, Stratix, and Cyclone devices.

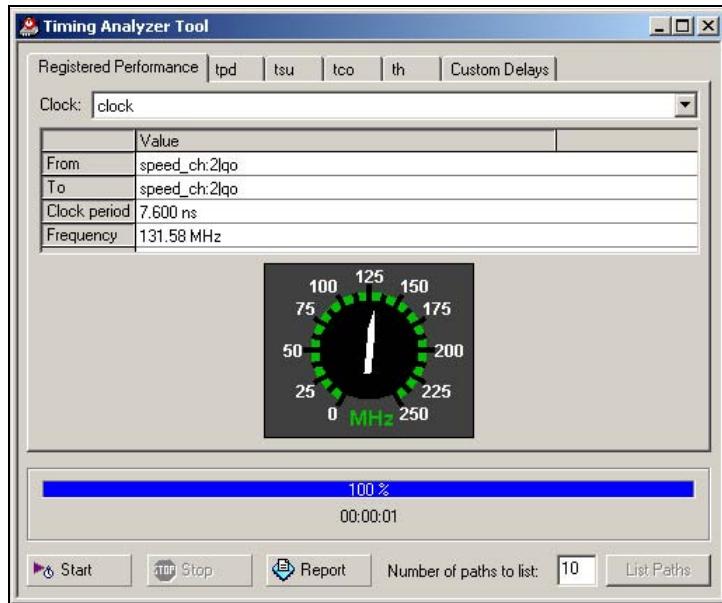
To further reduce compilation times, turn on **Limit to one fitting attempt** in the **Fitter Settings** page in the **Settings** dialog box (Assignments menu).

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 (Assignments menu). The value of the seed does not affect compilation time or the fitter effort level. It simply provides a different starting point for the fitter algorithm.

Timing Analysis

You can use the Quartus II Timing Analyzer to analyze more complex clocking schemes than is possible with the MAX+PLUS II Timing Analyzer.

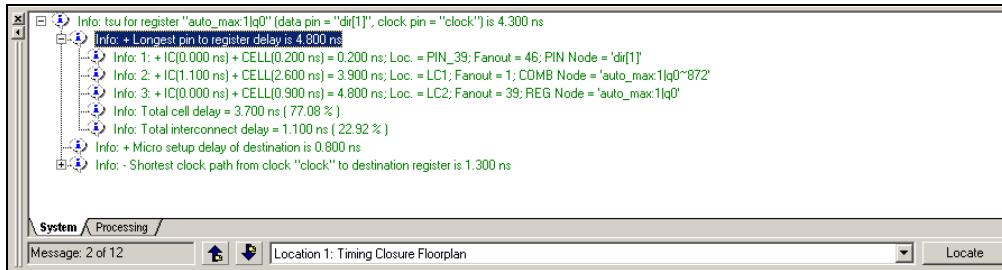
Launch the Timing Analyzer Tool by choosing **Timing Analyzer** (MAX+PLUS II menu) or by choosing **Timing Analyzer Tool** (Tools menu). See [Figure 1-17](#). To start the analysis, click **Start** in the Timing Analyzer Tool or choose **Start > Start Timing Analyzer** (Processing menu).

Figure 1–17. Registered Performance Tab of the Timing Analyzer Tool

The Quartus II Timing Analyzer analyzes all clock domains in your design, including paths that cross clock domains. You can ignore paths that cross clock domains by creating a **Cut Timing Path** assignment or by turning on **Cut paths between unrelated clock domains** in the **Timing Requirements & Options** page in the **Settings** dialog box (Assignments menu).

You can view the results by clicking on the available tabs or by clicking **Report** in the Timing Analyzer Tool. The Quartus II Timing Analyzer reports both f_{MAX} and slack. Slack is the margin by which a 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 Timing Analyzer Tool and click **List Paths**. This displays a detailed description of the path in the **System** tab of the **Messages** window (see [Figure 1–18](#)).

Figure 1–18. Messages Window Displaying Detailed Timing Information

For more information, see the *Quartus II Timing Analysis* chapter in Volume 3 of the *Quartus II Handbook*.

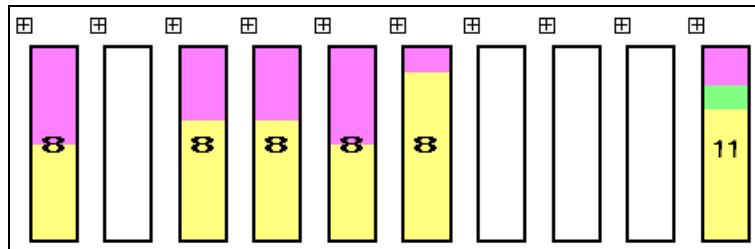
Timing Closure Floorplan

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 it easy to improve your design performance.

To view the Timing Closure Floorplan, choose **Floorplan Editor** (MAX+PLUS II menu) or **Timing Closure Floorplan** (Assignments menu).

The Timing Closure Floorplan Editor provides Package (Top and Bottom) and Interior Cell views equivalent to the MAX+PLUS II Device and LAB views. In addition to these views, available from the View menu, you can also choose from the Interior MegaLABs (where applicable), Interior LABs, and Field views.

The Interior LABs view hides cell (logic cell, Adaptive Logic Module (ALM), and macrocell) details and shows LAB information (see [Figure 1–19](#)). You can display the number of cells used in each LAB by selecting **Show Usage Numbers** (View menu).

Figure 1–19. 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 for your design.

The View Critical Paths feature shows a percentage of all critical paths in your floorplan. You can enable this feature by choosing **Show Critical Paths** (View menu). You can control the number of critical paths shown by modifying the settings in the **Critical Paths Settings** dialog box (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.

 You can show lower level details in any view by right-clicking on a resource and choosing **Show Details** (right button pop-up menu).



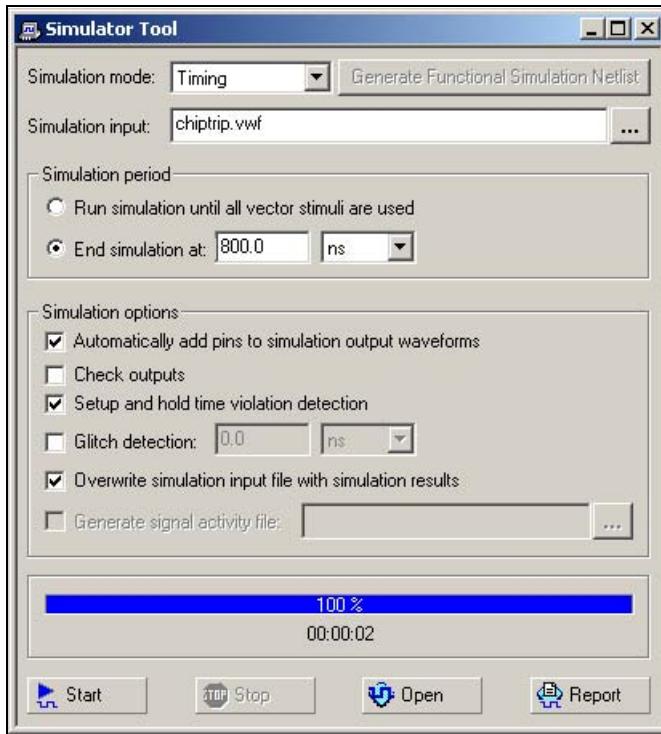
For more information, see the *Timing Closure Floorplan* chapter in Volume 2 of the *Quartus II Handbook*.

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 provides an easy-to-use integrated solution. It uses the compiler database to simulate the logical and timing performance of your design (Figure 1–20). When performing timing simulation, the simulator uses place-and-route timing information.

Figure 1–20. Quartus II Simulator Tool

You can use Vector Table Output Files (.tbl), Vector Waveform Files (.vWF), Vector Files (.vec), or an existing SCF 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(s) as shown in [Table 1-3](#).

<i>Table 1-3. Altera Timing Simulation Library Files</i>	
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 selecting the tool under **Tool name** on the **EDA Tool Settings > Simulation** page of the **Settings** dialog box (Assignments menu).

You can generate a timing netlist for the selected EDA simulator tool by running a full compile or by choosing **Start > Start EDA Netlist Writer** (Processing menu). 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 Web Site 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 VCD file generated from a third-party simulation.



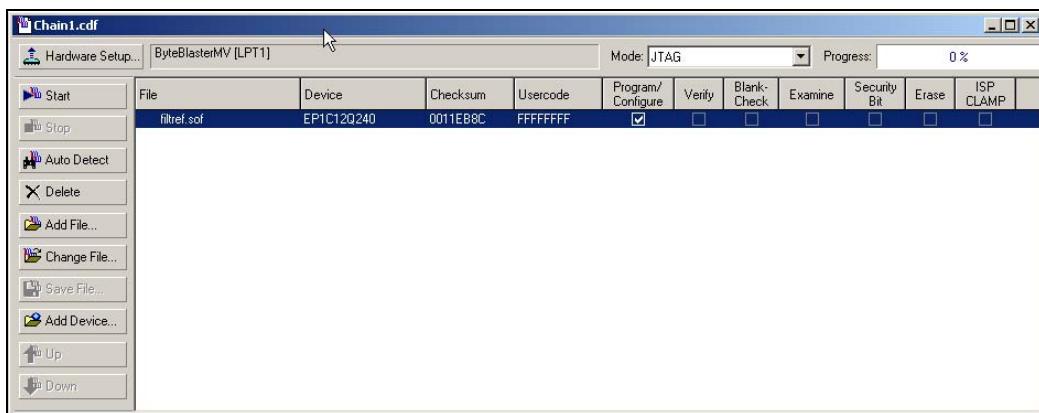
For more information on early power estimation, see the *Early Power Estimation* chapter in Volume 3 of the *Quartus II Handbook*. For more information about how to use the Power Analyzer tool, see the *Power Analysis* chapter in Volume 3 of the *Quartus II Handbook*.

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 (see [Figure 1–21](#)).

Click **Add File** or **Add Device** in the Programmer window to add a file or device, respectively.

Figure 1–21. Programmer Window



[Figure 1–21](#) 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 MAX+PLUS II Quick Menu and the Quartus II Quick Menu change according to the window that is active (see Figures 1–22 and 1–23). In the following example, the Graphic Editor window is active.

Figure 1–22. MAX+PLUS II Quick Menu

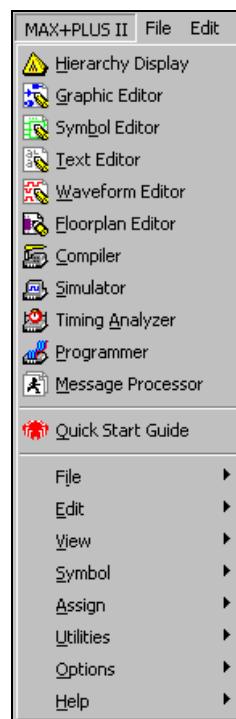
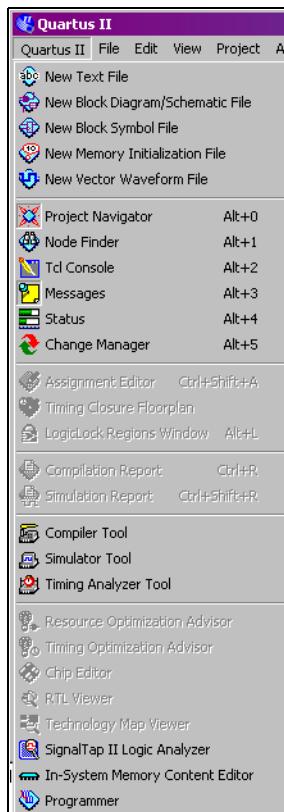


Figure 1–23. Quartus II Quick Menu



Quartus II Command Reference for MAX+PLUS II Users

Table 1–4. 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 1–4. Quartus II Command Reference for MAX+PLUS II Users (Part 1 of 11)	
MAX+PLUS II Software	Quartus II Software
MAX+PLUS II Menu	
 Hierarchy Display	 View > Utility Windows > Project Navigator
 Graphic Editor	 Block Editor
 Symbol Editor	 Block Symbol Editor
 Text Editor	 Text Editor
 Waveform Editor	 Waveform Editor
 Floorplan Editor	 Assignments > Timing Closure Floorplan
 Compiler	 Tools > Compiler Tool
 Simulator	 Tools > Simulator Tool
 Timing Analyzer	 Tools > Timing Analyzer Tool
 Programmer	 Tools > Programmer
 Message Processor	 View > Utility Windows > Messages
File Menu	
 File > Project > Name (Ctrl+J)	 File > Open Project (Ctrl+J)
 File > Project > Set Project to Current File (Ctrl+Shift+J)	 Project > Set as Top-Level Entity (Ctrl+Shift+J) or  File > New Project Wizard
 File > Project > Save & Check (Ctrl+K)	 Processing > Start > Start Analysis & Synthesis (Ctrl+K) or  Processing > Start > Start Analysis & Elaboration

**Table 1–4. Quartus II Command Reference for MAX+PLUS II Users
(Part 2 of 11)**

MAX+PLUS II Software	Quartus II Software
 File > Project > Save & Compile (Ctrl+L)	 Processing > Start Compilation (Ctrl+L)
 File > Project > Save & Simulate (Ctrl+Shift+L)	 Processing > Start Simulation (Ctrl+I)
File > Project > Save, Compile & Simulate (Ctrl+Shift+K)	Processing > Start Compilation & Simulation (Ctrl+Shift+K)
File > Project > Archive	Project > Archive Project
File > Project > <Recent Projects>	File > <Recent Projects>
File > Delete File	NA
File > Retrieve	NA
File > Info (Ctrl+I)	File > File Properties
File > Create Default Symbol	File > Create/Update > Create Symbol Files for Current File
File > Edit Symbol	(Block Editor) Edit > Edit Selected Symbol
File > Create Default Include File	File > Create/Update > Create AHDL Include Files for Current File
 File > Hierarchy Project Top (Ctrl+T)	 Project > Hierarchy > Project Top (Ctrl+T)
File > Hierarchy > Up (Ctrl+U)	 Project > Hierarchy > Up (Ctrl+U)
File > Hierarchy > Down (Ctrl+D)	 Project > Hierarchy > Down (Ctrl+D)
File > Hierarchy > Top	NA
 File > Hierarchy > Project Top (Ctrl+T)	 Project > Hierarchy > Project Top (Ctrl+T)
File > MegaWizard Plug-In Manager	 Tools > MegaWizard Plug-In Manager
(Graphic Editor) File > Size	NA
(Waveform Editor) File > End Time	(Waveform Editor) Edit > End Time
(Waveform Editor) File > Compare	 (Waveform Editor) View > Compare to Waveforms in File
(Waveform Editor) File > Import Vector File	 File > Open (Ctrl+O)
(Waveform Editor) File > Create Table File	File > Save As

**Table 1–4. Quartus II Command Reference for MAX+PLUS II Users
(Part 3 of 11)**

MAX+PLUS II Software	Quartus II Software
(Hierarchy Display) File > Select Hierarchy	NA
(Hierarchy Display) File > Open Editor	(Project Navigator) Double-click
(Hierarchy Display) File > Close Editor	NA
(Hierarchy Display) File > Change File Type	(Project Navigator) Select file in Files tab and choose Properties on right click menu
(Hierarchy Display) File > Print Selected Files	NA
(Programmer) File > Select Programming File	 File > Open
(Programmer) File > Save Programming Data As	 File > Save
(Programmer) File > Inputs/Outputs	NA
(Programmer) File > Convert SRAM Object Files	File > Convert Programming Files
(Programmer) File > Archive JTAG Programming Files	NA
(Programmer) File > Create Jam or SVF File	File > 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 > Compilation Report - Save Current Report on right click menu in Timing Analyzer sections
(Simulator) Create Table File	(Waveform Editor) File > Save As
(Simulator) Execute Command File	NA
(Simulator) Inputs/Outputs	NA
Edit Menu	
(Waveform Editor) Edit > Overwrite	(Waveform Editor) Edit > Value
(Waveform Editor) Edit > Insert	(Waveform Editor) Edit > Insert Waveform Interval

**Table 1–4. Quartus II Command Reference for MAX+PLUS II Users
(Part 4 of 11)**

MAX+PLUS II Software	Quartus II Software
(Waveform Editor) Edit > Align to Grid (Ctrl+Y)	NA
(Waveform Editor) Edit > Repeat	(Waveform Editor) Edit > Repeat Paste
(Waveform Editor) Edit > Grow or Shrink	Edit > Grow or Shrink (Ctrl+Alt+G)
(Text Editor) Edit > Insert Page Break	 (Text Editor) Edit > Insert Page Break
 (Text Editor) Edit > Increase Indent (F2)	 (Text Editor) Edit > Increase Indent
 (Text Editor) Edit > Decrease Indent (F3)	 (Text Editor) Edit > Decrease Indent
 (Graphic Editor) Edit > Toggle Connection Dot (Double-Click)	(Block Editor) Edit > Toggle Connection Dot
 (Graphic Editor) Edit > Flip Horizontal	 (Block Editor) Edit > Flip Horizontal
 (Graphic Editor) Edit > Flip Vertical	 (Block Editor) Edit > Flip Vertical
(Graphic Editor) Edit > Rotate	 (Block Editor) Edit > Rotate by Degrees
View Menu	
 View > Fit in Window (Ctrl+W)	 View > Fit in Window (Ctrl+W)
 View > Zoom In (Ctrl+Space)	 View > Zoom In (Ctrl+Space)
 View > Zoom Out (Ctrl+Shift+Space)	 View > Zoom Out (Ctrl+Shift+Space)
View > Normal Size (Ctrl+1)	NA
View > Maximum Size (Ctrl+2)	NA
(Hierarchy Display) View > Auto Fit in Window	NA
(Waveform Editor) View > Time Range	 View > Zoom
Assign Menu	
Assign > Device	 Assignments > Device
	or
	 Assignments > Settings (Ctrl+Shift+E)

**Table 1–4. Quartus II Command Reference for MAX+PLUS II Users
(Part 5 of 11)**

MAX+PLUS II Software	Quartus II Software
Assign > Pin/Location/Chip	Assignments > Assignment Editor - Locations category
Assign > Timing Requirements	Assignments > Assignment Editor - Timing category
Assign > Clique	Assignments > Assignment Editor - Cliques category
Assign > Logic Options	Assignments > Assignment Editor - Logic Options category
Assign > Probe	NA
Assign > Connected Pins	Assignments > Assignment Editor - Simulation category
Assign > Local Routing	Assignments > Assignment Editor - Local Routing category
Assign > Global Project Device Options	Assignments > Device - Device & Pin Options
Assign > Global Project Parameters	Assignments > Settings - Analysis & Synthesis - Default Parameters
Assign > Global Project Timing Requirements	Assignments > Timing Settings
Assign > Global Project Logic Synthesis	Assignments > Settings - Analysis & Synthesis
Assign > Ignore Project Assignments	Assignments > Assignment Editor - disable
Assign > Clear Project Assignments	Assignments > Remove Assignments
Assign > Back-Annotate Project	Assignments > Back-Annotate Assignments
Assign > Convert Obsolete Assignment Format	NA
Utilities Menu	
Utilities > Find Text (Ctrl+F)	Edit > Find (Ctrl+F)
Utilities > Find Node in Design File (Ctrl+B)	Project > Locate > Locate in Design File
Utilities > Find Node in Floorplan	Project > Locate > Locate in Timing Closure Floorplan
Utilities > Find Clique in Floorplan	NA
Utilities > Find Node Source (Ctrl+Shift+S)	NA

**Table 1–4. Quartus II Command Reference for MAX+PLUS II Users
(Part 6 of 11)**

MAX+PLUS II Software	Quartus II Software
Utilities > Find Node Destination (Ctrl+Shift+D)	NA
Utilities > Find Next (Ctrl+N)	 Edit > Find Next (F3)
Utilities > Find Previous (Ctrl+Shift+N)	NA
Utilities > Find Last Edit	NA
 Utilities > Search and Replace (Ctrl+R)	 Edit > Replace (Ctrl+H)
Utilities > Timing Analysis Source (Ctrl+Alt+S)	NA
Utilities > Timing Analysis Destination (Ctrl+Alt+D)	NA
Utilities > Timing Analysis Cutoff (Ctrl+Alt+C)	NA
Utilities > Analyze Timing	NA
Utilities > Clear All Timing Analysis Tags	NA
(Text Editor) Utilities > Go To (Ctrl+G)	 Edit > Go To (Ctrl+G)
(Text Editor) Utilities > Find Matching Delimiter (Ctrl+M)	 (Text Editor) Edit > Find Matching Delimiter (Ctrl+M)
(Waveform Editor) Utilities > Find Next Transition (Right Arrow)	(Waveform Editor) View > Next Transition (Right Arrow)
(Waveform Editor) Utilities > Find Previous Transition (Left Arrow)	(Waveform Editor) View > Next Transition (Left Arrow)
Options Menu	
Options > User Libraries	 Assignments > Settings (Ctrl+Shift+E)
Options > Color Palette	Tools > Options
Options > License Setup	Tools > License Setup
Options > Preferences	Tools > Options
(Hierarchy Display) Options > Orientation	NA
(Hierarchy Display) Options > Compact Display	NA
(Hierarchy Display) Options > Show All Hierarchy Branches	(Project Navigator) Expand All on right click menu

**Table 1–4. Quartus II Command Reference for MAX+PLUS II Users
(Part 7 of 11)**

MAX+PLUS II Software	Quartus II Software
(Hierarchy Display) Options > Hide All Hierarchy Branches	NA
(Editors) Options > Font	Tools > Options
(Editors) Options > Text Size	Tools > Options
(Graphic Editor) Options > Line Style	Edit > Line
 (Graphic Editor) Options > Rubberbanding	 Tools > Options
(Graphic Editor) Options > Show Parameters	 View > Show Parameter Assignments
(Graphic Editor) Options > Show Probes	NA
(Graphic Editor) Options > Show Pins/Locations/Chips	 View > Show Pin and Location Assignments
(Graphic Editor) Options > Show Clique, Timing & Local Routing Assignments	NA
(Graphic Editor) Options > Show Logic Options	NA
 (Graphic Editor) Options > Show All (Ctrl+Shift+M)	NA
(Graphic Editor) Options > Show Guidelines (Ctrl+Shift+G)	Tools > Options - Block/Symbol Editor page
(Graphic Editor) Options > Guideline Spacing	Tools > Options - Block/Symbol Editor page
(Symbol Editors) Options > Snap to Grid	Tools > Options - Block/Symbol Editor page
(Text Editor) Options > Tab Stops	Tools > Options - Text Editor page
(Text Editor) Options > Auto-Indent	Tools > Options - Text Editor page
(Text Editor) Options > Syntax Coloring	NA
(Waveform Editor) Options > Snap to Grid	 View > Snap to Grid
(Waveform Editor) Options > Show Grid (Ctrl+Shift+G)	Tools > Options - Waveform Editor page
(Waveform Editor) Options > Grid Size	Edit > Grid Size - Waveform Editor page

**Table 1–4. Quartus II Command Reference for MAX+PLUS II Users
(Part 8 of 11)**

MAX+PLUS II Software	Quartus II Software
(Floorplan Editor) Options > Routing Statistics	NA
 (Floorplan Editor) Options > Show Node Fan-In	 View > Routing > Show Fan-In
 (Floorplan Editor) Options > Show Node Fan-Out	 View > Routing > Show Fan-Out
 (Floorplan Editor) Options > Show Path	 View > Routing > Show Paths between Nodes
(Floorplan Editor) Options > Show Moved Nodes in Gray	NA
(Simulator) Options > Breakpoint	Processing > Simulation Debug > Breakpoints
(Simulator) Options > Hardware Setup	NA
(Timing Analyzer) Options > Time Restrictions	 Assignments > Timing Settings
(Timing Analyzer) Options > Auto-Recalculate	NA
(Timing Analyzer) Options > Cell Width	NA
(Timing Analyzer) Options > Cut Off I/O Pin Feedback	 Assignments > Timing Settings
(Timing Analyzer) Options > Cut Off Clear & Reset Paths	 Assignments > Timing Settings
(Timing Analyzer) Options > Cut Off Read During Write Paths	 Assignments > Timing Settings
(Timing Analyzer) Options > List Only Longest Path	NA
(Programmer) Options > Sound	NA
(Programmer) Options > Programming Options	Tools > Options - Programmer page
(Programmer) Options > Select Device	(Programmer) Edit > Change Device
(Programmer) Options > Hardware Setup	(Programmer) Edit > Hardware Setup
Symbol (Graphic Editor)	
Symbol > Enter Symbol (Double-Click)	 (Block Editor) Edit > Insert Symbol (Double-Click)

**Table 1–4. Quartus II Command Reference for MAX+PLUS II Users
(Part 9 of 11)**

MAX+PLUS II Software	Quartus II Software
Symbol > Update Symbol	Edit > Update Symbol or Block
Symbol > Edit Ports/Parameters	Edit > Properties
Element (Symbol Editor)	
Element > Enter Pinstub	Double-click on edge of symbol
Element > Enter Parameters	NA
Templates (Text Editor)	
Templates	(Text Editor) Edit > Insert Template
Node (Waveform Editor)	
Node > Insert Node (Double-Click)	Edit > Insert Node or Bus (Double-Click)
Node > Enter Nodes from SNF	Edit > Insert Node - click on Node Finder...
Node > Edit Node	Double-click on the Node
Node > Enter Group	Edit > Group
Node > Ungroup	Edit > Ungroup
Node > Sort Names	Edit > Sort
Node > Enter Separator	NA
Layout (Floorplan Editor)	
Layout > Full Screen	View > Full Screen (Ctrl+Alt+Space)
Layout > Report File Equation Viewer	View > Equations
Layout > Device View (Double-Click)	View > Package Top or View > Package Bottom
Layout > LAB View (Double-Click)	View > Interior Labs
Layout > Current Assignments Floorplan	View > Assignments > Show User Assignments
Layout > Last Compilation Floorplan	View > Assignments > Show Fitter Assignments
Processing (Compiler)	
Processing > Design Doctor	Processing > Start > Start Design Assistant

**Table 1–4. Quartus II Command Reference for MAX+PLUS II Users
(Part 10 of 11)**

MAX+PLUS II Software	Quartus II Software
Processing > Design Doctor Settings	 Assignments > Settings - Design Assistant
Processing > Functional SNF Extractor	Processing > Generate Functional Simulation Netlist
Processing > Timing SNF Extractor	 Processing > Start Analysis & Synthesis
Processing > Optimize Timing SNF	NA
Processing > Linked SNF Extractor	NA
Processing > Fitter Settings	 Assignments > Settings - Fitter Settings
Processing > Report File Settings	 Assignments > Settings
Processing > Generate AHDL TDO File	NA
Processing > Smart Recompile	 Assignments > Settings - Compilation Process
Processing > Total Recompile	 Assignments > Settings - Compilation Process
Processing > Preserve All Node Name Synonyms	 Assignments > Settings - Compilation Process
Interfaces (Compiler)	 Assignments > EDA Tool Settings
Initialize (Simulator)	
Initialize > Initialize Nodes/Groups	NA
Initialize > Initialize Memory	NA
Initialize > Save Initialization As	NA
Initialize > Restore Initialization	NA
Initialize > Reset to Initial SNF Values	NA
Node (Timing Analyzer)	
Node > Timing Analysis Source (Ctrl+Alt+S)	NA
Node > Timing Analysis Destination (Ctrl+Alt+D)	NA
Node > Timing Analysis Cutoff (Ctrl+Alt+C)	NA

**Table 1–4. Quartus II Command Reference for MAX+PLUS II Users
(Part 11 of 11)**

MAX+PLUS II Software	Quartus II Software
Analysis (Timing Analyzer)	
Analysis > Delay Matrix	(Timing Analyzer Tool) Delay tab
Analysis > Setup/Hold Matrix	NA
Analysis > Registered Performance	(Timing Analyzer Tool) Registered Performance tab
JTAG (Programmer)	
JTAG > Multi-Device JTAG Chain	(Programmer) Mode: JTAG
JTAG > Multi-Device JTAG Chain Setup	(Programmer) Window
JTAG > Save JCF	File > Save
JTAG > Restore JCF	File > Open
JTAG > Initiate Configuration from Configuration Device	Tools > Options - Programmer page
FLEX (Programmer)	
FLEX > Multi-Device FLEX Chain	(Programmer) Mode: Passive Serial
FLEX > Multi-Device FLEX Chain Setup	(Programmer) Window
FLEX > Save FCF	File > Save
FLEX > Restore FCF	File > Open

qii51004-2.2

Introduction

This chapter includes Quartus® II Support for both HardCopy® II and HardCopy series devices. This chapter is divided into the following sections:

- Quartus II Support for HardCopy II Devices
- Quartus II Support for HardCopy Stratix® & HardCopy APEX™ Devices

HardCopy II Device Support

The Altera® HardCopy II device family in 1.2-V, 90-nm process technology provides a powerful 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 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.

The HardCopy II device family is Altera's third generation of HardCopy structured ASICs. Previous families of HardCopy devices include 0.13- μ m HardCopy Stratix and 0.18- μ m HardCopy APEX™ devices. These families offer low-cost solutions to the Stratix and APEX FPGA devices for volume production.



For more information on HardCopy II, HardCopy Stratix, and HardCopy APEX families, refer to the respective sections for these families 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 of your project.

Quartus II Features for HardCopy II Planning

Quartus II software provides the capability to design your HardCopy II device using a Stratix II device as a prototype. Beginning with version 4.2, Quartus II software contains two features for HardCopy II device planning:

- **HardCopy II Device Resource Guide**—Helps select the best HardCopy II device for migration by comparing the resources required for a design with the resources available in the various HardCopy II devices.
- **Migration Devices** dialog box—Identifies compatible Stratix II and HardCopy II devices for migration. This feature constrains the pins of your Stratix II FPGA prototype to be compatible with your HardCopy II device.

HardCopy II Prototyping Flow

The flow for developing a HardCopy II prototype is very similar to the flow for developing any other design for a Stratix II FPGA. The difference is that you must also specify which HardCopy II device the design eventually migrates to. The flowchart in [Figure 2-1](#) provides an overview highlighting the process for specifying a HardCopy II device.

To prototype your HardCopy II design in the Quartus II software, you must first select the Stratix II device family in the **Device** category of the **Settings** dialog box (Assignments menu).

Once you compile your Stratix II design successfully, you can view the HardCopy II Device Resource Guide in the Fitter report to evaluate which HardCopy II devices meet your design's resource requirements. You may see that a small change in the design allows it to fit a smaller, less expensive device. If so, you can make the change and repeat the process. When you are satisfied with the compilation results and the choice of Stratix II and HardCopy II devices, select a HardCopy II device in the **Migration Devices** dialog box and recompile.

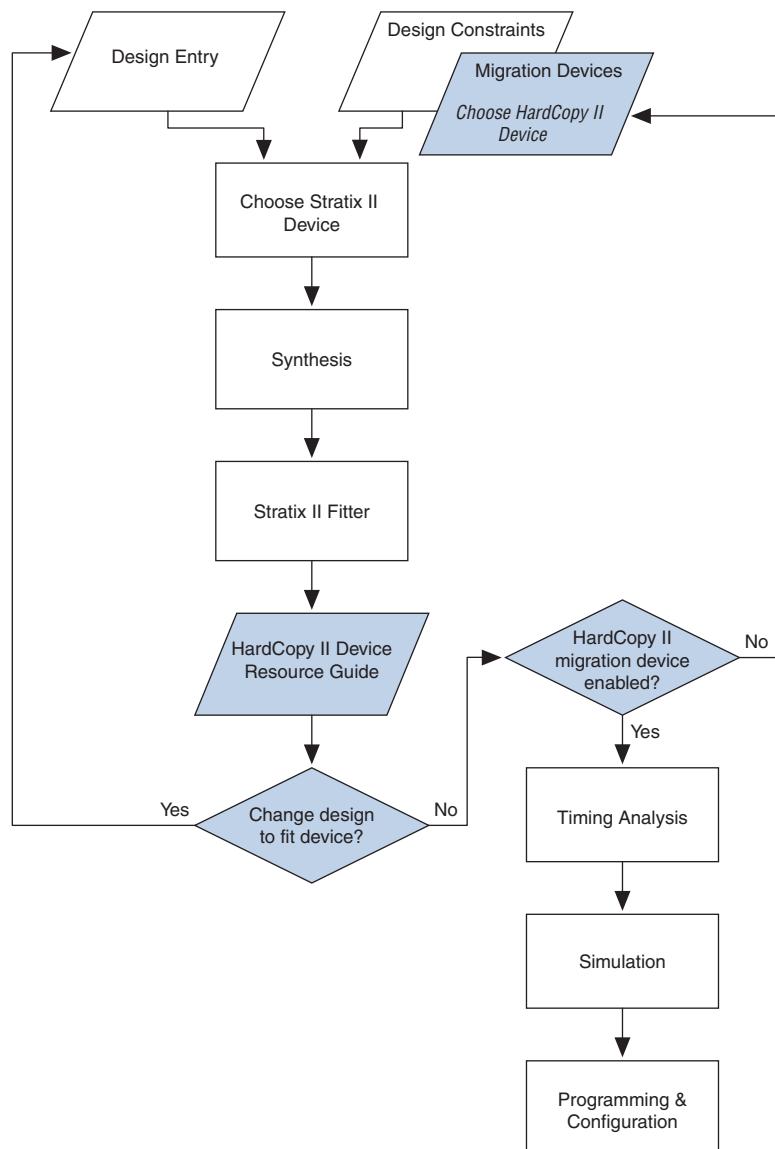
From here the flow continues normally with timing analysis and optimization of the design, simulation and verification of the design, and, finally, programming and configuring the Stratix II FPGA.



For more information on the prototyping process for HardCopy II devices, refer to the *Prototyping Strategy for HardCopy II Devices* chapter of the *HardCopy Series Handbook*.



For more information on the overall design flow using Quartus II software, see the *Introduction to Quartus II* manual in the **Products > Literature > Quartus II** section of the Altera web site (www.altera.com).

Figure 2–1. HardCopy II Device Prototype Flow

HardCopy II Device Resource Guide

The HardCopy II Device Resource Guide compares the resources required for a successfully compiled design to the resources available in the various HardCopy II devices. The Guide rates each HardCopy II device and each device resource for how well it fits the design. The HardCopy II Device Resource Guide is generated for all designs successfully compiled for Stratix II devices, and is found in the Fitter folder of the Compilation Report. Figure 2-2 shows an example of the HardCopy II Device Resource Guide. The color code is explained in Table 2-1.

Figure 2-2. 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										
	Resource	Stratix II EP2S130	HC210W [*]	HC210	HC220	HC220	HC230	HC240	HC240	
1	Migration Compatibility			None	None	None	Medium	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	
4	Logic	--	40%	22%	22%	15%	10%	10%		
5	-- Logic cells	35695 ALUTs		--	--	--	--	--	--	
6	-- DSP elements	0		--	--	--	--	--	--	
7	Pins									
8	-- Total	515		515 / 335	515 / 493	515 / 495	515 / 699	515 / 743	515 / 952	
9	-- Differential Input	0		0 / 70	0 / 90	0 / 90	0 / 128	0 / 224	0 / 272	
10	-- Differential Output	0		0 / 50	0 / 70	0 / 70	0 / 112	0 / 200	0 / 256	
11	-- PCI / PCI-X	0		0 / 167	0 / 245	0 / 247	0 / 359	0 / 367	0 / 472	
12	-- DQ	0		0 / 20	0 / 44	0 / 44	0 / 178	0 / 178	0 / 178	
13	-- DQS	0		0 / 8	0 / 18	0 / 18	0 / 72	0 / 72	0 / 72	
14	Memory									
15	-- M-RAM	6		6 / 0	6 / 2	6 / 2	6 / 6	6 / 9	6 / 9	
16	-- M4K blocks & M512 blocks	44		44 / 190	44 / 408	44 / 408	44 / 609	44 / 768	44 / 768	
17	PLLs									
18	-- Enhanced	2		2 / 2	2 / 2	2 / 2	2 / 4	2 / 4	2 / 4	
19	-- Fast	0		0 / 2	0 / 2	0 / 2	0 / 4	0 / 8	0 / 8	
20	DLLs	0		0 / 1	0 / 1	0 / 1	0 / 2	0 / 2	0 / 2	
21	SERDES									
22	-- RX	0		0 / 19	0 / 31	0 / 31	0 / 44	0 / 92	0 / 116	
23	-- TX	0		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	
26	-- ASMI	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	
28	-- JTAG	0		0 / 1	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1	

* Device is preliminary. Overall performance is expected to be degraded.

Use this report to determine which HardCopy II device is a potential candidate for migration of your Stratix II design. The HardCopy II package must be compatible with the Stratix II device. A logic resource

Table 2–1. 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 fit with target device migration enabled in the Migration Devices 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.
Orange (Medium)	The design can migrate to the Hardcopy II package. However, the design has not been fit with target device migration enabled in the Migration Devices 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. Consult your Product Field Applications Engineer for a recommended course of action.
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 for Table 2–1:

- (1) The package resource is constrained by the Stratix II FPGA that the design was compiled for. Only vertical migration devices within the same package are able to migrate to HardCopy II devices.

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.

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 and DSP blocks. The DSP blocks in HardCopy II devices match the functionality, performance, and timing of the Stratix II DSP blocks. The M4K and M-RAM memory blocks in HardCopy II devices are equivalent to the Stratix II memory blocks.



For more information on the HardCopy II device resources, see the *Introduction to HardCopy II Devices* and the *Description, Architecture & Features* chapters in the *HardCopy II Device Family Data Sheet* in the *HardCopy Series Handbook*.

The report example in [Figure 2–2](#) 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. Since the HC230F1020 device was not specified as a migration target during the compilation, its package and migration compatibility are rated orange or Medium. The migration compatibility 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 FineLine BGA HC240 device is rated red because it is only compatible with the Stratix II EP2S180F1020 device. The selection

of the HC210 and HC220 devices is further restricted due to the total number of pins used and the number of M-RAM memory blocks required.

To use an HC220F780 device for this example design, reduce the total number of I/O pins used and the number of M-RAM blocks implemented, and change the Stratix II FPGA to a compatible device such as the EP2S130F780 device.

[Figure 2–3](#) shows the report after the (unchanged) design was recompiled with the HardCopy II HC230F1020 device specified as a migration target. Now the HC230F1020 device's package and migration compatibility are rated green or High.

Figure 2–3. HardCopy II Device Resource Guide with Target Migration Enabled

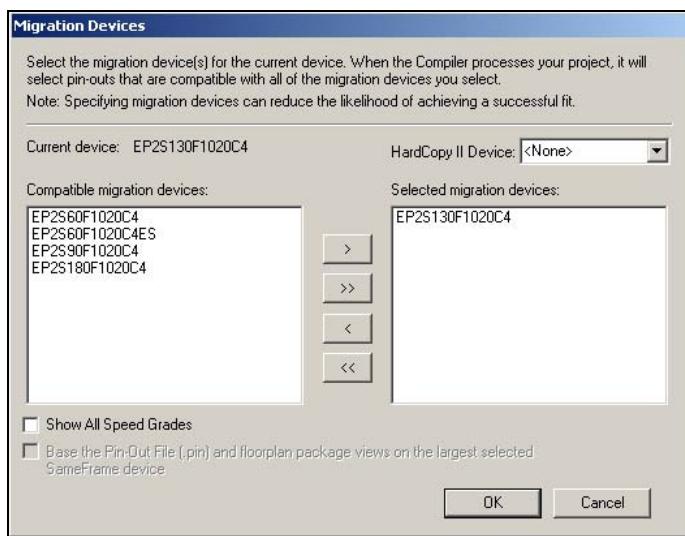
HardCopy II Device Resource Guide									
Color Legend:									
-- Green: -- Package Resource: The Hardcopy II package can be migrated from the Stratix II FPGA selected package, and the design has been fitted with									
	Resource	Stratix II EP2S130	HC210w*	HC210	HC220	HC220	HC230	HC240	HC240
1	Migration Compatibility			None	None	None	High	None	None
2	Primary Migration Constraint			Package	Package	Package		Package	Package
3	Package	FBGA - 1020	FBGA - 484	FBGA - 484	FBGA - 672	FBGA - 780	FBGA - 1020	FBGA - 1020	FBGA - 1508

Migration Devices Dialog Box

The **Migration Devices** dialog box is part of the Quartus II design constraints. It constrains the pin assignments for your Stratix II device to also fit your migration targets. Use the **Migration Devices** dialog box to select the target HardCopy II device after you have compiled the Stratix II prototype design and reviewed the HardCopy II Device Resource Guide. The **Migration Devices** dialog box is found in the **Device** category of the **Settings** dialog box (Assignments menu).

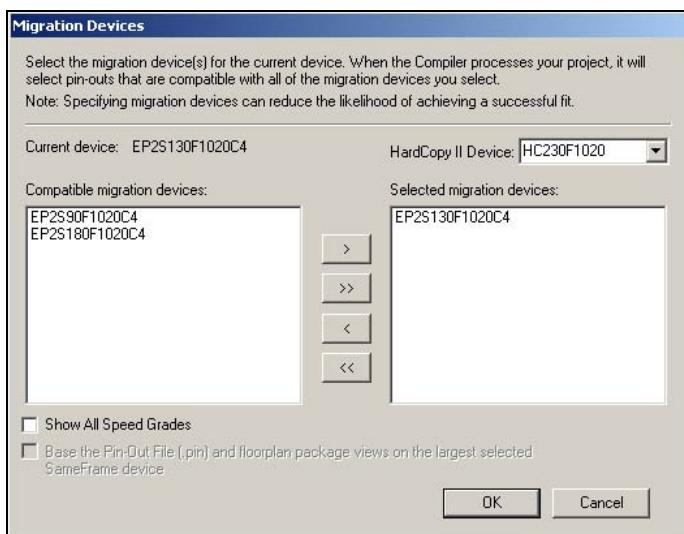
After the first time you compile a Stratix II design with the Quartus II software, the **Migration Devices** dialog box displays only those Stratix II devices that are migration-compatible with your current selected Stratix II device, as shown in [Figure 2–4](#). Select a HardCopy II device from the **HardCopy II Device** list.

Figure 2–4. Migration Devices Dialog Box Showing Compatible Stratix II Devices



Once you have selected the HardCopy II device from the **HardCopy II Device** list, the Quartus II software filters the FPGAs in the **Compatible migration devices** list to show only those FPGA devices that can be used to prototype the selected HardCopy II device, as shown in [Figure 2–5](#). The Quartus II software does not filter this list based on compiled information about your design, so your design may not fit in some of the Stratix II devices listed. You can select more than one FPGA, but only one HardCopy II device, for migration.

Figure 2–5. Migration Devices Dialog Box Showing Selected HardCopy II Device & Compatible Stratix II Devices



For example, a design compiled for a Stratix II EP2S130F1020 device with vertical package migration selected can migrate to an EP2S60F1020, EP2S90F1020, or EP2S180F1020 device. As a prototype, the design can migrate to the HardCopy II HC230F1020 device, which is selected from the **HardCopy II Device** list, as shown in Figure 2–5. After you have selected a HardCopy II device, the **Compatible migration devices** list is restricted to the Stratix II devices supported by the HardCopy II device. After the HC230F1020 device is selected, the only allowable migration devices are the EP2S90F1020, EP2S180F1020, and the (already selected) EP2S130F1020 devices.

Once you have targeted a HardCopy II device and selected any additional Stratix II devices for prototyping from the **Compatible Migration Devices** list in the **Migration Devices** dialog box, re-compile your design and proceed with your prototype optimization and verification.

For more information on the steps to migrate your design from your Stratix II FPGA prototype to a HardCopy II device, contact your Altera Product Field Applications Engineer.

Conclusion

Beginning with version 4.2, Altera's Quartus II software allows you to design for HardCopy II devices and to develop prototypes using Stratix II FPGAs. This is done using the standard development process with the addition of the HardCopy II Device Resource Guide and the **Migration Devices** dialog box. After successfully compiling a design, the HardCopy II Device Resource Guide helps you select a suitable HardCopy II device by comparing the resources required by the design to the resources available in the various HardCopy II devices. The **Migration Devices** dialog box is then used to specify the HardCopy II device and constrain the pin assignments to be compatible with both the HardCopy II device and the Stratix II prototype. You can then recompile and continue with the usual optimization and verification processes. The resulting Stratix II FPGA is a fully functional prototype for the HardCopy II device.

HardCopy Stratix & HardCopy APEX Device Support

The 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 or 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 and power 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 chapter 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

The Quartus II software version 4.2 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**

The Quartus II software version 4.2 determines routing and associated buffer insertion for HardCopy Stratix designs, and provides the Timing Analyzer with more accurate information on the delays than was possible in previous versions of the Quartus II software. The Quartus II Archive (.qar) 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 enables 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.

- **HardCopy Stratix Power Estimator**

This power estimation tool is launched from the Quartus II software to estimate power consumed by the HardCopy Stratix devices.



The HardCopy APEX Power Estimator is available on the Altera web site (www.altera.com) in the **Products > Devices > HardCopy APEX > Design Utilities > HardCopy APEX Power Estimator**.

HARDCOPY_FPGA_PROTOTYPE, HardCopy Stratix, & Stratix Devices

You must use the HARDCOPY_FPGA_PROTOTYPE virtual devices available in 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 APEX device family.

Table 2–2 compares HARDCOPY_FPGA_PROTOTYPE devices, Stratix devices, and HardCopy Stratix devices.

Table 2–2. Qualitative Comparison of HARDCOPY_FPGA_PROTOTYPE to Stratix & HardCopy Stratix Devices		
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
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 2–3 lists the resources available in each of the HardCopy Stratix devices.

Table 2–3. HardCopy Stratix Device Physical Resources								
Device	LEs	ASIC Equivalent gates (K)⁽¹⁾	M512 Blocks	M4K Blocks	M-RAM Blocks	DSP Blocks	PLLs	Max. 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 2–3:

- (1) Combinatorial and registered logic do not include digital signal processing (DSP) blocks, on-chip RAM, or phase-locked loop (PLL)s.
- (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 on HardCopy Stratix devices, see 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.



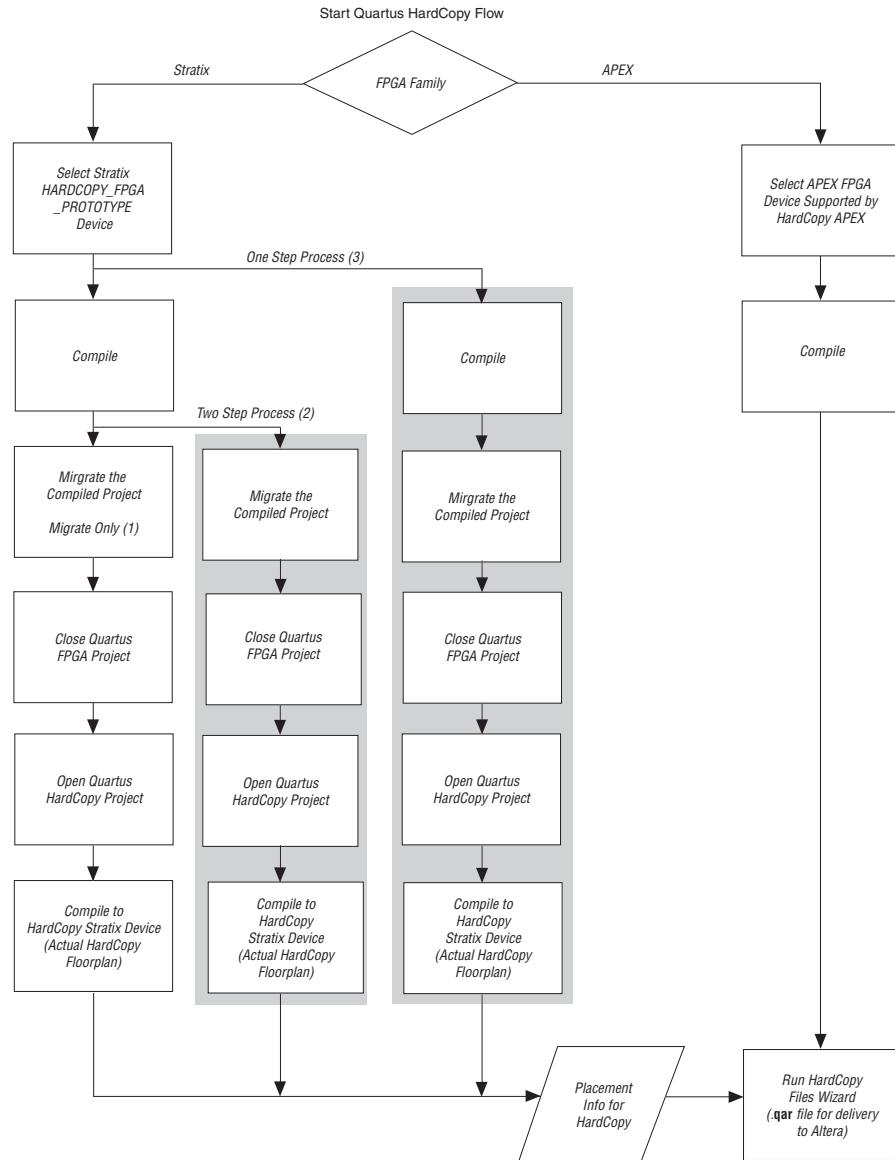
For more information on the .sof file and programming Stratix FPGA devices, see the *Programming and Configuration* chapter of the *Introduction to Quartus II Manual*.

HardCopy Design Flow

Figure 2–6 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, see “[HardCopy Timing Optimization Wizard Summary Page](#)” on page 2–21 and “[Generating the HardCopy Design Database](#)” on page 2–31.

Figure 2–6. HardCopy Stratix & HardCopy APEX Design Flow Diagram**Notes for Figure 2–6:**

- (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 2–6](#).

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.

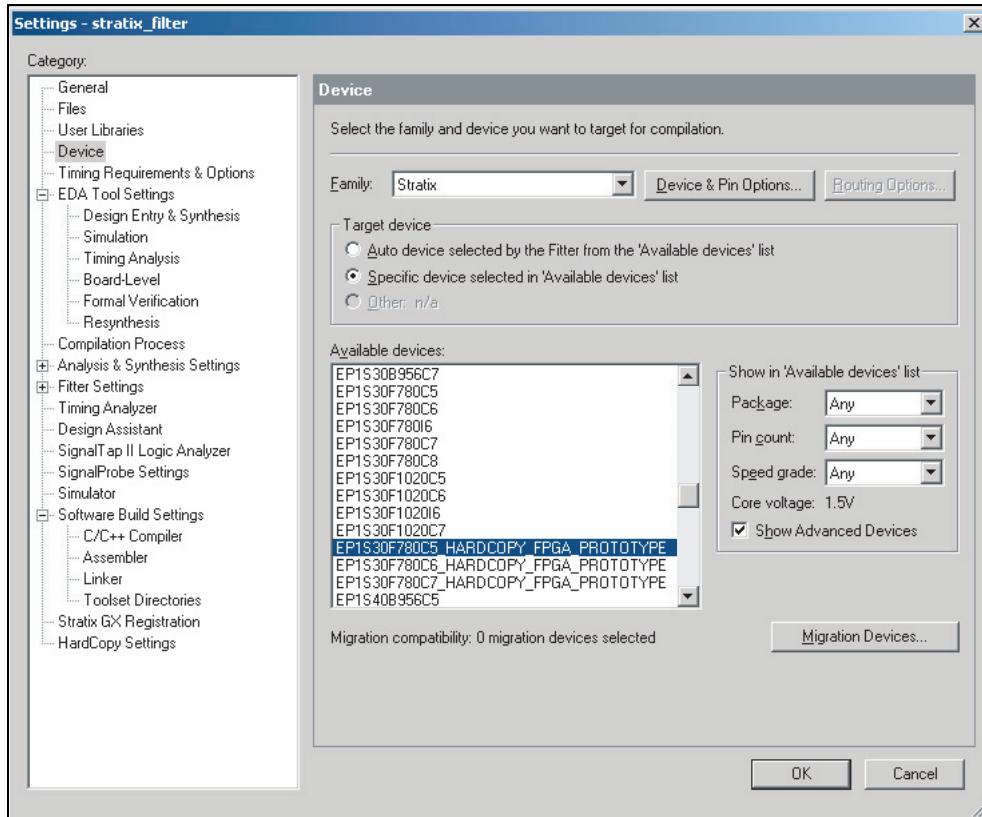
How to Design HardCopy Stratix Devices

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. Select **Device** (Assignments menu), then select **Stratix** in the **Family** list. Select the desired HARDCOPY_FPGA_PROTOTYPE device in the **Available Devices** list, as shown in [Figure 2–7](#).

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.

Figure 2–7. Selecting a HARDCOPY_FPGA_PROTOTYPE Device

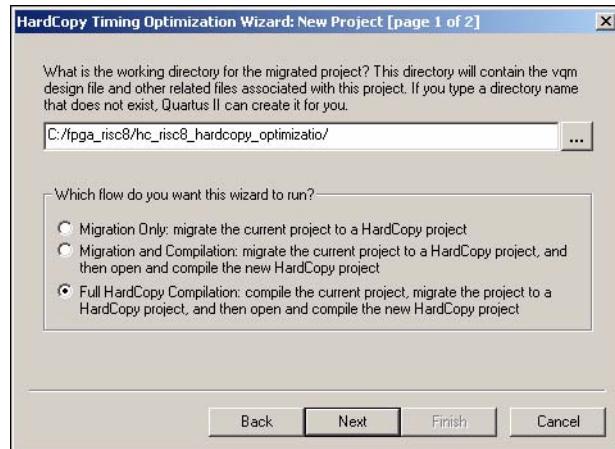
3. Choose **Settings** (Assignments menu). In the **Category** list select **HardCopy Settings** and specify the input transition timing to be modeled for both clock and data input pins. These transition times are used in static timing analysis during back-end timing closure of the HardCopy device.
4. Add constraints to your HARDCOPY_FPGA_PROTOTYPE device and compile the design by choosing **Start Compilation** (Processing menu).

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, choose the HardCopy Timing Optimization Wizard in the **HardCopy Utilities** (Project menu).

At this point, you are presented with the following three choices to target the designs to HardCopy Stratix devices, as shown in [Figure 2–8](#).

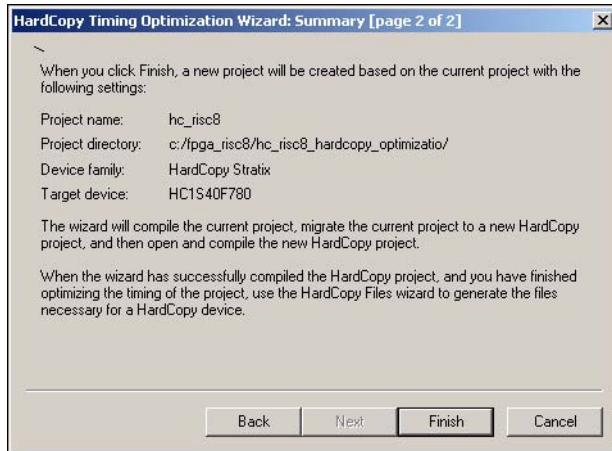
- **Migration Only**—You can now perform the following tasks manually to target the design to a HardCopy Stratix device. See “[Performance Estimation](#)” on page 2–22 if you need more information on how to perform these tasks.
 - Close the existing project
 - Open the migrated HardCopy Stratix project
 - 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 2–8. 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 2–9](#).

Figure 2–9. HardCopy Timing Optimization Wizard Summary Page



When either of the second two options in [Figure 2–8](#) 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, see [“Performance Estimation” on page 2–22](#). 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 existing 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 & 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

The Quartus II software version 4.2 supports HardCopy Stratix design optimization by providing floorplans for placement optimization and HardCopy Stratix timing models. These features enable 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 2–10 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.

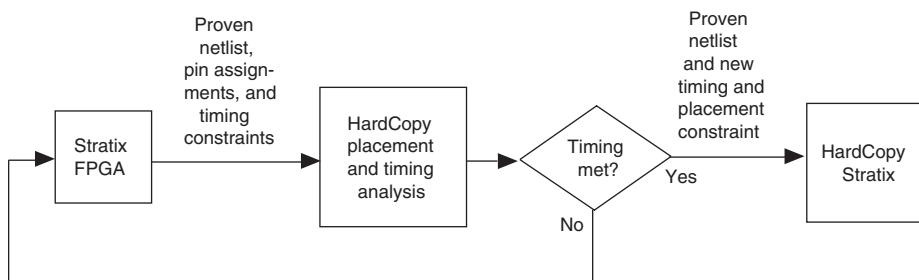
or

- 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 their equivalent -6 speed grade Stratix FPGA device. These performance numbers are highly design dependent, and you must obtain final performance numbers from Altera.

Figure 2-10. 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_PROTOTYPE device.
2. Choose **HardCopy Utilities > HardCopy Timing Optimization Wizard** (Project menu).
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 timing analysis 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 2–4](#).

Table 2–4. 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.



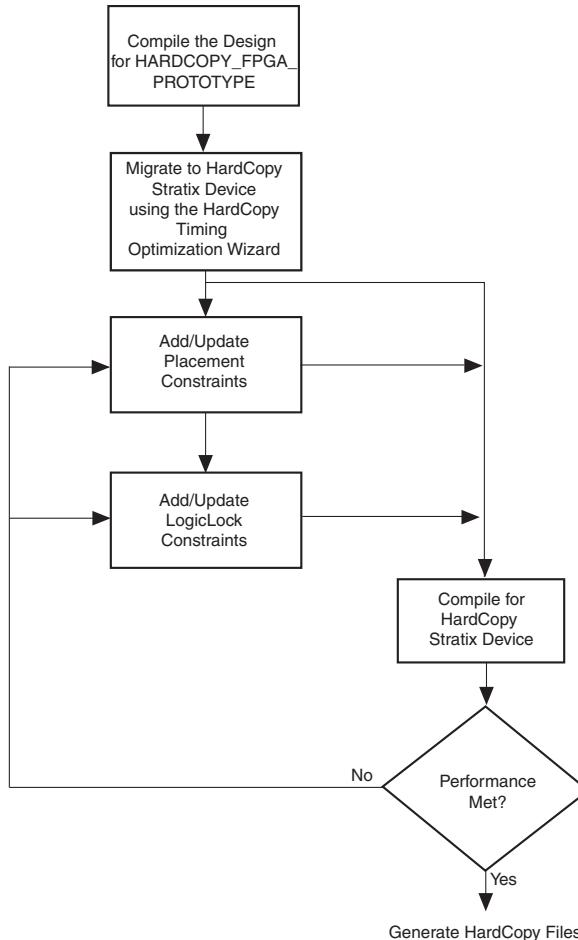
Performance estimation is not supported for HardCopy APEX devices in the Quartus II software. You can optimize your design by modifying the RTL code or the FPGA design and the constraints. You should contact Altera to discuss any desired performance improvements with HardCopy APEX devices.

Buffer Insertion

The Quartus II software version 4.2 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 .qar file. This buffer insertion improves the estimation of the Quartus II Timing Analyzer for the HardCopy Stratix device.

Placement Constraints

The Quartus II software version 4.2 supports placement constraints and LogicLock regions for HardCopy Stratix devices. [Figure 2–11](#) shows an iterative process to modify the placement constraints until the best placement for the HardCopy Stratix device is achieved.

Figure 2–11. Placement Constraints Flow for HardCopy Stratix Devices

Location Constraints

This section provides information on 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, see the *LogicLock Design Methodology* 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.

The following are two examples of LogicLock assignments.

LogicLock Region Definition in the HARDCOPY_FPGA_PROTOTYPE.qsf 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
```

LogicLock Region Definition in the Migrated HardCopy Stratix .qsf 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 cannot 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 on design recommendations and HDL coding styles, see 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 have at least 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 of the **Settings** dialog box (Assignments menu) prior to running the design. In this dialog box, you can choose whether to run the 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, choose **Start > Start Design Assistant** (Processing menu).

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, see 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 .qar file is generated at the same directory level as the targeted project, either before or after optimization. [Table 2–5](#) shows the archive directory structure and files collected by the HardCopy Files Wizard.

Table 2–5. 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

```



The Design Assistant automatically runs when the HardCopy Files Wizard is started.

After creating the migration database with the HardCopy Timing Optimization Wizard, you must compile the design before generating the project archive. You receive an error if you create the archive before compiling the design.

Static Timing Analysis (STA)

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>.hcpy_v.sdo`—Standard Delay Format Output File (SDF)
- `<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 on static timing analysis, see the *Quartus II Timing Analysis* and the *Synopsys PrimeTime Support* chapters in Volume 3 of the *Quartus II Handbook*.

Power Estimation

The Quartus II software has built-in capability for estimating HardCopy Stratix device power consumption by evaluating the following design components:

- Target device and package
- Temperature grade
- Clock domain f_{MAX}
- Device resources used

HardCopy Stratix Power Estimator

The HardCopy Stratix 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 on simulation-based power estimation, see the *Power Estimation & Analysis* Section in Volume 3 of the *Quartus II Handbook*.

Opening the HardCopy Stratix Power Estimator

The HardCopy Stratix Power Estimator page on the Altera web site is opened in the Quartus II software. The Quartus II software automatically fills in the necessary information when you open the page. You can modify the values and estimate the estimated power consumed under various conditions.

The estimator can also be opened independently of the Quartus II software on the Altera web site (www.altera.com) by clicking on **Products > Devices**. Select **HardCopy Stratix**, then click on the **Power Estimator** link.



You must enter design-specific information manually if you run the estimator directly.

To estimate HardCopy Stratix power consumption, follow these steps:

1. After compiling the design for a HardCopy Stratix device, choose **HardCopy Utilities > HardCopy Power Estimation** (Project menu), and click **OK**.
2. Enter values for the following variables in the spreadsheet and click **Calculate** to get the total power (P_{TOTAL}).
 - Average number of logic elements
 - Average capacitive load
 - DC output power
 - Ambient temperature

For more information on power estimation, see the Quartus II Help.



The HardCopy Stratix Power Estimator is run from the Quartus II software when the target is still HARDCOPY_FPGA_PROTOTYPE device. However, power is estimated for the HardCopy Stratix device, not for the FPGA.

Use the Stratix FPGA Power Estimator to estimate power consumption for the HARDCOPY_FPGA_PROTOTYPE.



On average, HardCopy Stratix devices are expected to consume 40% less power than the equivalent FPGA.

HardCopy APEX Power Estimator

The HardCopy APEX Power Estimator is also a web-based estimator that can be run from the Altera web site (www.altera.com) in the **Products > Devices > HardCopy APEX > Design Utilities > HardCopy APEX Power Estimator**. You cannot open this feature in the Quartus II software.

With the HardCopy APEX Power Estimator, you can estimate the power consumed by HardCopy APEX devices and design systems with the appropriate power budget.



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, see the *Tcl Scripting* chapter in Volume 2 of the *Quartus II Handbook*.

Targeting Designs to HardCopy APEX Devices

The Quartus II software version 4.2 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 2–6](#) shows the files collected for HardCopy APEX by the HardCopy Files Wizard.

Table 2–6. HardCopy APEX Files Collected by the HardCopy Files Wizard

<pre> <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 </pre>
--

See “[Generating the HardCopy Design Database](#)” on page 2–31 for information on 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 of your design in a HardCopy device. You should contact Altera for more information on 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.

qii51005-2.1

Introduction

A major benefit of programmable logic is that it accommodates changes to the 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 when engineering development begins or when all system elements are being integrated.

These last-minute design changes are commonly referred to as engineering change orders (ECOs). ECOs are defined as small changes to the functionality of a design, after the design has been fully compiled, i.e., synthesis and place-and-route are completed.

ECOs are usually intended to correct errors found in the programmable logic design during debugging, or after changes that are made to the design specification to compensate for design problems in other areas of the system design. The operation of the system design cannot easily be changed in these areas.

As the project nears completion, a significant amount of time and effort has been invested in achieving timing closure in the programmable logic device (PLD). It is crucial that the programmable logic design flow is optimized to support ECOs in an efficient manner.

Impact of Last Minute Design Changes

ECOs have an impact on the following areas of a system design:

- Performance
- Compile time
- Verification
- Documentation

Performance

When a small change is made to the design functionality, it can result in previous design optimizations being lost. Typical examples of design optimizations are floorplan optimizations and physical synthesis. Ideally, there should be a means to preserve the design optimizations that have already been made. This will focus future optimizations that might be made to the design on the areas of the design to which the ECO changes were applied.

Compile Time

In the traditional programmable logic design flow, a small change in the design results in a complete recompilation of the design, i.e., synthesis and place-and-route. Thus, the process of making small changes to the design to reach the final implementation on a board can be a very long process. Ideally, to reach the desired functionality and to reach timing closure, a small change in functionality should result in a reduced compilation time. This can be achieved using incremental compilation technology that uses the previous fit information on the areas of the design that have not been affected by the ECOs.

Verification

After any design change, the impact of the change on the design must be verified. This verification is achieved through timing analysis and simulation. You can choose to limit the verification to the area of the design that is impacted by the ECOs. This is accomplished by running timing analyses on select paths and having the option to perform simulation on gate level and timing simulation netlists.

Documentation

Changes to the project files must be tracked. This helps other users reproduce the results at a later date. Ideally, you should be able to have multiple compilation revisions so that others can try out changes without corrupting the results that have been previously obtained.

ECO Support

ECOs can be applied at either of two stages of a typical design flow:

- HDL level
- Netlist level

Traditionally in programmable logic design, ECOs have been applied at the HDL level. This is because the tools to easily create ECOs and to enable design sign-off at the netlist level have generally not been available for PLDs.

ECO Support at the HDL Level

An ECO at the HDL level is a small incremental change to the design's Verilog or VHDL source. This change may range from a single line to several lines of code modified within a module or entity. Typical examples of such modifications are:

- Changes to the state encoding of a finite state machine
- Addition of pipeline registers to improve design performance
- Signal duplication to reduce fan-out
- Adding a term to a conditional expression
- Changing the polarity of register control signals

A few changes to the source code can produce many changes to the netlist produced by other EDA synthesis or tools such as the Quartus® II software's integrated synthesis. During the synthesis process, the synthesis tools generally preserve the names of registers from the HDL source code, but automatically generate names for the combinational (look-up table level) nodes. This automatic name generation is necessary to accommodate the synthesis optimization performed on the HDL source to use the target device resources more efficiently.

Thus, a minor source code change can result in many changes to the names in the synthesis netlist. The changes in the synthesis netlist can be due to either of the following reasons:

- The node names in the new netlist implement different functionality than in the previous netlist
- The node names in the new netlist implement the same functionality as in the previous netlist, but have different names

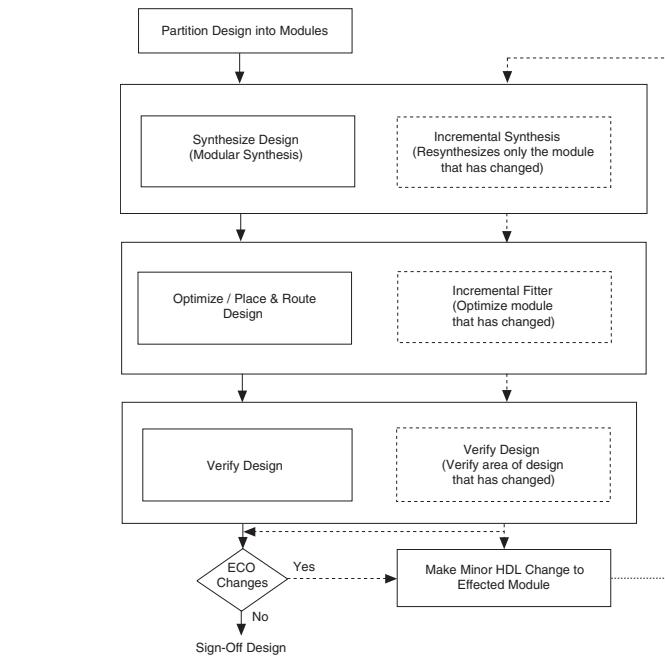
To leverage the previous design optimizations and to reduce the compilation time, there must be a means of performing an incremental compilation on the nodes with the new functionality and preserving the previous optimizations on the nodes that have not changed. Thus, a means of identifying nodes that maintain the same functionality but have different names is essential for providing an ECO flow that truly works. Such a solution is provided with the incremental fitting feature available in the Quartus II software.

The Quartus II software incremental fitting feature performs a comparison between the original synthesis netlist and the new netlist containing the ECO changes. It matches nodes based upon names, functionality, fan-in, and fan-out. Those nodes that can be matched inherit the assignments from the previous fit.

Thus, the incremental fitting feature can preserve existing fitting information and timing. This feature limits any timing and fitting changes to the logic that has changed in functionality and reduces the compilation time.

To limit the changes caused by ECOs, it is recommended that users adopt a modular design flow. A modular design flow, combined with the incremental compilation features mentioned previously, minimizes the changes in performance caused by ECOs and reduces compilation time. Partitioning the design to adopt a modular design flow facilitates the placing of each module in the floorplan for performance. The Quartus II software provides the LogicLock™ feature to optimize the floorplan of modular designs. The *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook* describes how to apply the LogicLock methodology to a modular design flow. [Figure 3–1](#) details the recommended design flow to support ECO changes at the HDL level.

Figure 3–1. Design Flow to Support ECO Changes



ECO Support at the Netlist Level

For certain ECO changes, it can be quicker to make changes at the netlist level rather than at the HDL level. This happens when you are debugging the design on silicon and need a very fast turnaround in generating a programming file for debugging the system.

A typical application occurs when you uncover a problem on the board and isolate the problem to the appropriate nodes or I/O cells on the PLD. You then need to be able to quickly correct the functionality of the offending logic cell or the properties of the I/O cell and generate a new programming file in minutes. In doing this, you can verify the operation of the change without having to modify the HDL and perform a synthesis and place-and-route operation. This minimizes the disruption to the board verification procedure.

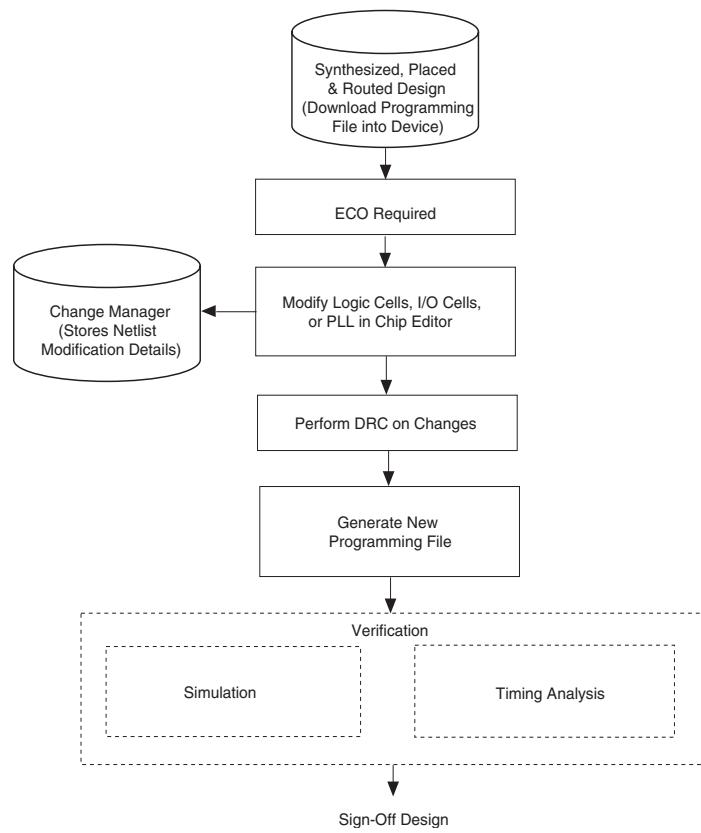
If this quick fix works, you do not need to change the HDL source code and rerun place-and-route. You should have the option to:

- Document the change that has been made
- Easily recreate the steps taken to produce the changes to the design
- Generate EDA simulation netlists for verification of the design
- Perform timing analysis on the design

These capabilities are provided in the Chip Editor feature of the Quartus II software.

The Quartus II Chip Editor allows you to make functional changes to individual logic cells and to the I/O cell and phase-locked loop (PLL) parameters. These changes are stored in the Quartus II Change Manager log. This allows you to control the application of the changes, and generate a tool command language (Tcl) file. This Tcl commands file recreates the changes on the original netlist, documents the changes made to the project, and enables you to recreate the changes on the original design files at a later date, without having to change the HDL source. You can regenerate an EDA simulation netlist for the modified design if it is necessary to perform a gate-level simulation of the modified design. If the designer needs to rerun timing analysis to sign-off the design, timing analysis can be rerun on the netlist containing the ECO changes.

[Figure 3-2](#) shows the flow for ECO changes at the netlist level.

Figure 3–2. Design Flow for ECO Changes at the Netlist Level

Conclusion

Support for ECOs requires a combination of a modular design methodology and the appropriate software design tools.

The Quartus II software provides you with the software tools and the design methodology to successfully perform ECOs at both the HDL and netlist level for programmable logic designs. This reduces the design cycle time and provides faster timing closure on designs that require last minute changes.



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 4, Design Recommendations for Altera Devices
- Chapter 5, Recommended HDL Coding Styles

Revision History

The table below shows the revision history for [Chapters 4](#) and [5](#).

Chapter(s)	Date / Version	Changes Made
5	Dec. 2004 v2.1	<p>Updated for Quartus II software version 4.2:</p> <ul style="list-style-type: none"> ● Chapter 5 was formerly Chapter 6 in version 4.1. ● General formatting and editing updates. ● Detailed what you must specify for the Quartus II Timing Analyzer to issues hardware requirements on page 4–2. ● Added reference for additional details about timing requirements and analysis to page 4–2. ● Minor rewording to the introductory paragraph on “Design Guidelines”. ● Added information about performing timing analysis on asynchronous ports to page 4–5. ● “Latches” was moved before “Delay Chains”. ● Updated description of FPGA devices in “Latches”. Added new information about inferred latches to page 4–5. ● Updated description of “Delay Chains” on page page 4–6. ● Updated Figure 4–2 through Figure 4–8. ● Added new note to “Clocking Schemes” that describes clock relationships, clock routing, and data transfers between clocks. ● Reworded description of “Ripple Counters”. ● Added two new paragraphs describing “Multiplexed Clocks” on page 4–11. ● Added new information about using dedicated hardware for clock gating to “Gated Clocks” on page 4–11. ● Reworded description of “Recommended Clock-Gating Method”. ● “Hierarchical Design Partitioning” was updated to include synthesis and incremental synthesis. New reference was added for more information on incremental synthesis flows. ● Rewrote introduction to “Clock Network Resources”. Additional text was added describing factors that affect the ability of the the Quartus II software to automatically use global routing. Support for global routing is clarified for older Altera device families.
	June 2004 v.2.0	<ul style="list-style-type: none"> ● Updates to tables, figures, and coding examples. ● New functionality for Quartus II software 4.1.
	Feb. 2004 v1.0	Initial release.

Chapter(s)	Date / Version	Changes Made
6	Dec. 2004 v2.1	<ul style="list-style-type: none">● Chapter 6 was formerly Chapter 7 in version 4.1.● Updates to tables and figures.● New functionality for Quartus II software 4.2.
	June 2004 v2.0	<ul style="list-style-type: none">● Updates to tables and figures.● Added and updated section for State Machines.● Update to Verilog HDL for State Machines.● New functionality for Quartus II software 4.1.
	Feb. 2004 v1.0	Initial release.

qii51006-2.1

Introduction

Today's 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 and reliability and faster time-to-market when designing with Altera® devices, you should:

- 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



For specific HDL coding examples and recommendations, including coding guidelines for targeting dedicated device hardware, such as memory and DSP blocks, see the *Recommended HDL Coding Styles* chapter in Volume 1 of the *Quartus II Handbook*. For information about migrating designs to HardCopy devices, see the *HardCopy Series Design Guidelines* chapter in the *HardCopy Series 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 consistently meet your design goals. 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, if you plan to migrate your design to a high-volume solution such as Altera HardCopy devices, or if you are prototyping an ASIC, then synchronous design practices help ensure successful migration.

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 your clock frequencies and other timing requirements, the Quartus® II 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.



In order 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 time.



For details about timing requirements and analysis in the Quartus II software, see the *Quartus II Timing Analysis* chapter 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 today’s 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 4-4](#). 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 hardware description language (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.

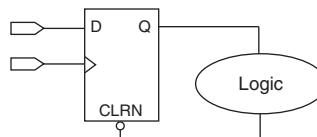
Combinational Logic Structures

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 (LE) or adaptive logic modules (ALM). In some cases when 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 4–1](#).

Figure 4–1. Combinational Loop through Asynchronous Control Pin





To perform timing analysis in the Quartus II software on asynchronous ports such as the `clear` or `reset`, click **More Settings** and turn on **Enable Recovery/Removal Analysis** from the **Timing Requirements & Option** page of the **Settings** dialog box (Assignments menu).

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, meaning 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 `1pm_latch` function ensures that the implementation will be glitch-free in Altera architectures. Some third-party synthesis tools list the number of `1pm_latch` functions that are inferred. When using Quartus II integrated synthesis, these latches are reported in a section of the Compilation Report called **User-Specified**

and Inferred Latches. 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 Quartus II software provides an option called **Analyze latches as synchronous elements** that 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 like 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 that you avoid using latches to ensure that you can completely analyze and verify the timing performance and reliability of your design.

Delay Chains

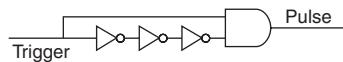
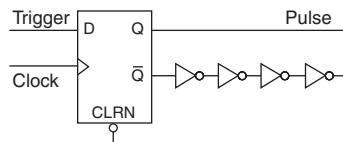
Delay chains occur when two or more consecutive nodes with a single fan-in and a single fan-out are used to cause delay. Inverters are often chained together to add delay. Delay chains are sometimes used to resolve race conditions created by other asynchronous design practices.

As discussed above, delays in PLD designs can change with each place-and-route cycle. Effects like 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 [4-3](#) 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 & 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 4-2](#). These techniques are purely asynchronous and therefore should be avoided.

Figure 4–2. Asynchronous Pulse Generators**Using an AND Gate****Using a Register**

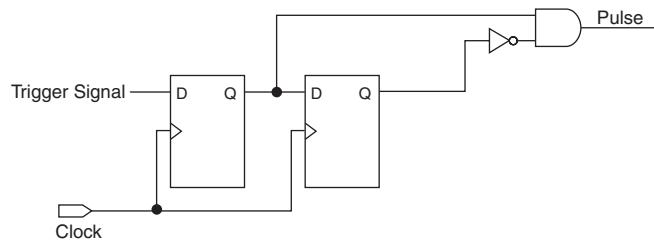
In “Using an AND gate” (Figure 4–2), a trigger signal feeds both inputs of a 2-input AND gate, but the design inverts or adds a delay chain to one of the inputs. The width of the pulse depends on the relative delays of the path that feeds the gate directly and the one that goes through the delay. This 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 4–2), 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 need to use a pulse generator, use synchronous techniques, as shown in Figure 4–3.

Figure 4–3. 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 where 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 4–15 for a detailed explanation.

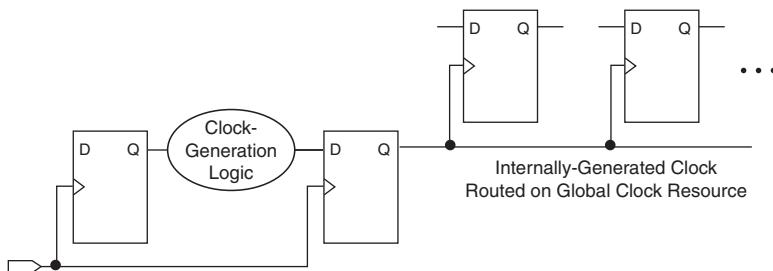
Avoid data transfers between different clocks wherever possible. If a data transfer between different clocks is needed, use FIFO circuitry. You can use the clock uncertainty features in the Quartus II software to compensate for the variable delays between clock domains. Consider setting a Clock Setup 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 4-4](#).

Figure 4-4. 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 need to 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 4–9, 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.

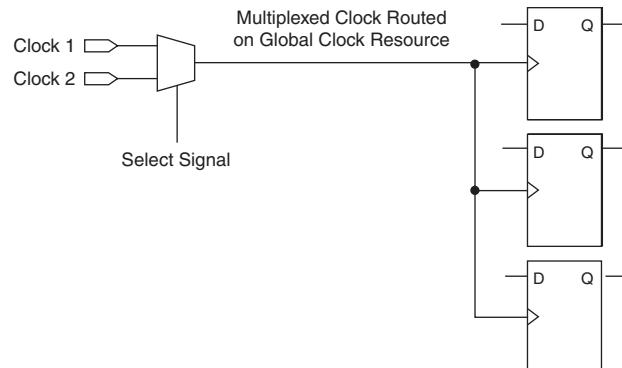
Ripple Counters

Altera recommends avoiding ripple counters in your design to simplify verification. 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.

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 4–5](#). For example, telecommunications applications that deal with multiple frequency standards often use multiplexed clocks.

Figure 4–5. Multiplexing Logic & Clock Sources



Adding multiplexing logic to the clock signal can create the problems discussed 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, then 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 analyses 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.

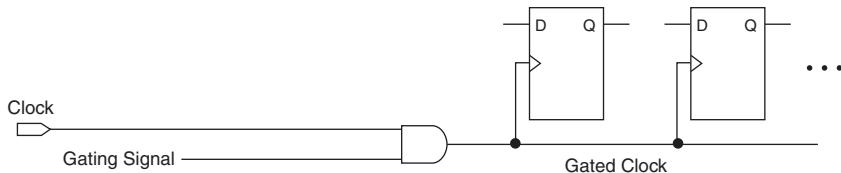
Altera recommends using dedicated hardware to perform clock multiplexing when it is available, instead of using multiplexing logic. For example, you can use the Clock Switchover feature of the PLL in the Stratix® series of devices, or the Clock Control Block in Stratix II and Cyclone™ II devices. These 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.



See the appropriate device data sheet or handbook for device-specific information on 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 4–6](#). When a clock is turned off, the corresponding clock domain is shut down and becomes functionally inactive.

Figure 4–6. Gated Clock

You can use gated clocks to reduce power consumption in some device architectures. 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 using dedicated hardware to perform clock gating, if it is available in your target device, rather than using multiplexing logic. For example, you can use the clock control block in Stratix II and Cyclone II 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.



See the appropriate device data sheet or handbook for device-specific information on clocking structures.

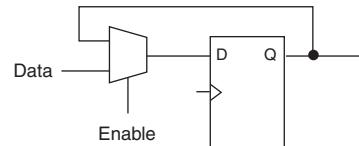
From a functional point of view, you can shut down a clock domain in a purely synchronous manner using a synchronous clock enable signal. However, when using a synchronous clock enable scheme, the clock network continues toggling. This practice does not reduce power consumption as much as gating the clock at the source does. In most cases, you should use a synchronous scheme such as those described in the “[Synchronous Clock Enables](#)” section. For improved power reduction when gating clocks with logic, see “[Recommended Clock-Gating Method](#)”.

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. See [Figure 4–7](#).

Figure 4–7. Synchronous Clock Enable

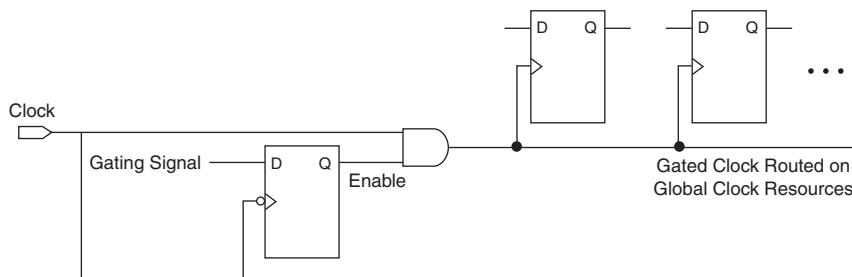


Recommended Clock-Gating Method

Only use gated clocks 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 4–8](#) 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, whenever possible, gate the clock at the source so that you can shut down the entire clock network instead of gating it further along the clock network at the registers.

Figure 4–8. Recommended Clock Gating Technique



In the technique shown in [Figure 4–8](#), 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 4–8](#)). Using this technique, only

one input of the gate that turns the clock on and off changes at a time that 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 4–8](#), 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.

Hierarchical Design Partitioning

A hierarchical design consists of multiple design blocks linked together in a hierarchy. When a design is partitioned hierarchically, you can synthesize, optimize and simulate the individual design blocks separately. You can use the LogicLock™ design flow to follow a block-based design methodology where each block is placed and routed independently, then all blocks in the hierarchy are combined at the top level. Some synthesis tools have features to help you create separate netlist files or maintain separate parts of a netlist file for different parts of your design, to support block-based design techniques or incremental synthesis. The Quartus II software also offers incremental synthesis, where different partitions of your design are synthesized separately from each other to help reduce resynthesis time and maintain synthesis node names for parts of the design that are not resynthesized.



For more information on the LogicLock design methodology, see the *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*. For more information on incremental synthesis flows in your synthesis tool, see the appropriate chapter in the *Synthesis* section in Volume 1 of the *Quartus II handbook*.

When using a hierarchical or incremental design methodology, it is important to consider how the design is partitioned to achieve good results. Altera recommends the following practices for partitioning designs:

- Partition the design at functional boundaries.
- Minimize the I/O connections between different partitions.
- Register all inputs and outputs of each block. This makes logic synchronous and avoids glitches and avoids any delay penalty on signals that cross between partitions. Registering I/Os typically eliminates the need to specify timing requirements for signals that connect between different blocks.
- Do not use “glue logic” or connection logic between hierarchical blocks. When you preserve hierarchy boundaries, glue logic is not merged with hierarchical blocks. Your synthesis software may optimize glue logic separately, which can degrade synthesis results and is not efficient when used with the LogicLock design methodology.
- Remember that logic is not synthesized or optimized across partition boundaries, which means any constant values (signals set to GND, for example) will not be propagated across partitions.
- Do not use tri-state signals or bidirectional ports on hierarchical boundaries. 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 Altera device. Because this requires optimizing through hierarchies, lower-level boundary tri-state signals are not supported with a block-level design methodology.
- Limit clocks to one per block. Partitioning the design into clock domains makes synthesis and timing analysis easier.
- Place state machines in separate blocks to speed optimization and provide greater encoding control.
- Separate timing-critical functions from non-timing-critical functions.
- Limit the critical timing path to one hierarchical block. You can group the logic from several design blocks to ensure the critical path resides in one block.

Targeting Clock & 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 that you limit 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.

Today's 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 and low-skew clock networks in the FPGA device. The Quartus II software will automatically use global routing for high-fanout control signals, PLL outputs, and signals feeding the global clock pins on the device. You can make explicit **Global Signal** logic option settings using the **Assignment Editor** (Assignment Menu) when 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's 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 on the inherent priority order of register control signals in Altera device architecture, see 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 consistently meet your design goals. 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.

qii51007-2.1

Introduction

Your hardware description language (HDL) coding style 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 knowledge of the design, and synthesis tools cannot always know the design intent. You are often in the best position to improve your quality of results.

This chapter discusses coding style recommendations to ensure optimal synthesis results when targeting Altera® devices. This chapter provides code examples for inferring Altera megafunctions from HDL code and targeting certain functions in Altera device architectures, along with device-specific coding recommendations for certain types of logic and some general coding guidelines.



For more general guidelines on structuring your design, refer to the *Design Recommendations for Altera Devices* chapter in Volume 1 of the *Quartus II Handbook*.

Instantiating & Inferring 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 functions may offer more efficient logic synthesis and device implementation. You can scale the megafunction's size by simply setting parameters.

Megafunctions include the library of parameterized modules (LPM) and Altera device-specific megafunctions.



You must use megafunctions to access some Altera device-specific features, such as memory, digital signal processing (DSP) blocks, low-voltage differential signal (LVDS) drivers, phase-locked loops (PLLs), transceivers, and double data rate input and output (DDIO) circuitry.

Altera megafunctions are easy to instantiate and offer efficient device implementation. Some designers, however, prefer to make their code independent of device family or vendor, and prefer not to instantiate megafunctions directly. In these cases, follow the guidelines in this chapter to ensure your HDL code infers the appropriate Altera

megafunction. In addition, for some designs, generic HDL code may provide better results than instantiating a megafunction. The following general guidelines provide some examples:

- 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 more complicated arithmetic such as synchronous loadable counters, LPM functions can give you access to detailed architecture-specific functionality that is difficult to infer from HDL code.
- 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™ devices, but use the LPM function only if you want to force a specific implementation.
- Avoid division operations where possible. Division is an inherently slow operation. Many designers use multiplications creatively to produce division results. If you must divide, the `LPM_DIVIDE` function provides the best results possible.

The following sections describe how to use megafunctions by instantiating them in your HDL code or inferring them from generic HDL code and includes a list of which megafunctions can be inferred from HDL code.

Instantiating Altera Megafunctions in HDL Code

If you decide to instantiate a megafunction in your HDL code, use one of the following methods:

- Use the Quartus® II software MegaWizard® Plug-In Manager to parameterize the function and create a wrapper file.
- Instantiate the function directly using the port and parameter definitions.

Instantiating Megafunctions Using the MegaWizard Plug-In Manager

Altera recommends that you use the **MegaWizard Plug-In Manager** (Tools menu) to instantiate megafunctions. The wizard provides a graphical 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 should be generated. The wizard generates an Altera HDL (AHDL), Verilog HDL, or VHDL wrapper file (depending on which language you choose on the

first page of the wizard) that instantiates the megafunction with the correct parameters, as well as other files including a Component Declaration File (.cmp) for VHDL and an Include File (.inc) for AHDL. You can then instantiate the wrapper file in your HDL code using the sample instantiation file `<output file>_inst.tdf/v/vhd`. See [Table 5–1](#) for a list of generated files.

When using certain megafunctions with synthesis tools outside the Quartus II software, you have the option of creating a clear box body instead of a wrapper file. The clear box netlist file is a fully synthesizable Altera megafunction, or LPM function, for use with third-party electronic design automation (EDA) synthesis tools. When implementing a megafunction with the clear box model, you provide the third-party synthesis tool with information about the architectural details used in the Quartus II software. This enables certain synthesis tools to better report timing and resource utilization estimates. In addition, the synthesis tool may be able to use the timing information to focus its timing-driven optimizations and improve the quality of results.



For information on clear box support 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 a clear box model, turn on **Generate a clear box body (for EDA tools only)** on the first page of the MegaWizard Plug-in Manager.

[Table 5–1](#) lists and describes the HDL and schematic files that the MegaWizard Plug-In Manager can generate.

Table 5–1. MegaWizard Plug-In Manager Generated Files (Part 1 of 2)

File	Description
<code><output file>.bsf</code>	Block Symbol File used in the Quartus II schematic editor
<code><output file>.cmp</code>	Component Declaration File used in VHDL designs
<code><output file>.inc</code>	Include File used in AHDL designs
<code><output file>.tdf (1)</code>	Megafunction wrapper file for instantiation in an AHDL design
<code><output file>.vhd (2) (4)</code>	Megafunction wrapper file, or clear box netlist file, for instantiation in a VHDL design
<code><output file>.v (3) (4)</code>	Megafunction wrapper file, or clear box netlist file, for instantiation in a Verilog HDL design
<code><output file>_bb.v (3)</code>	Hollow-body module declaration that can be used in Verilog HDL designs to specify port directions when black-boxing in third-party synthesis tools
<code><output file>_inst.tdf (2)</code>	Sample AHDL instantiation of the subdesign in the megafunction wrapper file
<code><output file>_inst.vhd (2)</code>	Sample VHDL instantiation of the entity in the megafunction wrapper file

Table 5–1. MegaWizard Plug-In Manager Generated Files (Part 2 of 2)

File	Description
<output file>_inst.v (4)	Sample Verilog HDL instantiation of the module in the megafunction wrapper file

Notes to Table 5–1:

- (1) The wizard generates this file only if you select AHDL output files.
- (2) The wizard generates this file only if you select VHDL output files.
- (3) The wizard generates this file only if you select Verilog HDL output files.
- (4) A megafunction wrapper file is created by default for most megafunctions. If you turn on the **Generate a clear box body (for EDA tools only)** option, the wizard creates a clear box netlist file which is used with third-party EDA synthesis tools. For more information about how to use the MegaWizard Plug-In Manager, see Quartus II Help.

Instantiating Megafunctions Using the Port & Parameter Definition

You can instantiate the megafunction directly in your AHDL, Verilog HDL, or VHDL code by calling the function like any other subdesign, module, or component.



See Quartus II Help for a list of the megafunction's 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 as described in the previous section for complex megafunctions such as PLLs, transceivers, and LVDS drivers.

Inferring Megafunctions from HDL Code

Synthesis tools, including Quartus II integrated synthesis, recognize certain types of HDL code and automatically infer the appropriate megafunction when a megafunction provides optimal results. That is, the software uses the Altera megafunction code when compiling your design—even though you did not specifically instantiate the megafunction. The software infers megafunctions resulting in 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 memory, DSP blocks, and shift registers—that generally provide improved performance compared with regular logic.

The following sections describe the types of logic that standard synthesis tools recognize and map to megafunctions. Synthesis software infers only the specific functions listed here. The software cannot infer other functions, such as PLLs, LVDS drivers, transceivers, or DDIO circuitry

from HDL code because these functions cannot be fully or efficiently described in HDL code. In some cases, synthesis tools provide options to turn off the inference of certain megafunctions.

Below is a list of these megafunctions that can be inferred by standard synthesis tools.



Certain megafunctions (such as `lpm_counter` and `lpm_addsub`) are used by Quartus II integrated synthesis while third party synthesis tools may use their own optimized versions of these logic functions.

The following sections describe the situations under which all these megafunctions can be inferred.

- `lpm_counter`
- `lpm_addsub`
- `lpm_mult`
- `altsyncram`
- `altmult_accum`
- `altmult_add`
- `lpm_ram_dp`
- `lpm_rom`
- `altshift_taps`



For features and options specific to a certain synthesis tool, refer to your synthesis tool's documentation or the appropriate chapter in the Synthesis section in Volume 1 of the *Quartus II Handbook*.

Inferring Counters from HDL Code

To infer counter functions, synthesis tools look for a set of registers that feed through a plus-one adder, a minus-one adder, or both. Quartus II integrated synthesis tools then converts the registers and logic to an `lpm_counter` megafunction, while third-party synthesis tools typically use their own optimized logic. If a design also has logic that implements control signals, the synthesis tool can recognize them as well. For example, the Quartus II software recognizes the following signals:

- Asynchronous clear
- Asynchronous set (to logic value 1)
- Asynchronous load
- Count enable
- Synchronous clear
- Synchronous set (to logic value 1)
- Synchronous load
- Clock enable

- Up/down

The following code samples show simple Verilog HDL and VHDL counter functions with different control signals. The Verilog HDL example includes count enable and asynchronous clear signals, and the VHDL example includes a synchronous load signal.

Verilog HDL Example of a Counter with Count Enable & Asynchronous Clear

```
module counter (clk, reset, result, ena);
    input clk;
    input reset;
    input ena;
    output [7:0] result;

    reg [7:0] result;

    always @ (posedge clk or posedge reset)
    begin
        if (reset)
            result = 0;
        else if (ena)
            result = result + 1;
    end
endmodule
```

VHDL Example of a Counter with Synchronous Load

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
ENTITY count IS
    PORT (
        clock: IN STD_LOGIC;
        sload: IN STD_LOGIC;
        data: IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        result: OUT STD_LOGIC_VECTOR (4 DOWNTO 0)
    );
END count;

ARCHITECTURE rtl OF count IS
    SIGNAL result_reg: STD_LOGIC_VECTOR (4 DOWNTO 0);
BEGIN
    PROCESS (clock)
    BEGIN
        IF (clock'event AND clock = '1') THEN
            IF (sload = '1') THEN
                result_reg <= data;
            ELSE
                result_reg <= UNSIGNED(result_reg) + 1;
            ENDIF;
        ENDIF;
    END PROCESS;
END;
```

```

        END IF;
    END IF;
END PROCESS;
result <= result_reg;
END rtl;

```

Inferring Adder/Subtractors from HDL Code

To infer adder and subtractor functions, synthesis tools look for adders and subtractors that have the same set of inputs and outputs that are multiplexed by a common signal. Quartus II integrated synthesis then merges the adders and subtractors and converts them to an `lpm_addsub` megafunction, while third-party synthesis tools typically use their own optimized logic.



Quartus II integrated synthesis does not infer `lpm_addsub` megafunctions when a third-party formal verification tool is selected under **EDA Tool Settings** in the **Settings** dialog box (Assignments menu).

The following code samples show Verilog HDL and VHDL examples of simple adder and subtractors. The VHDL example includes a small user-defined package to configure the widths.

Verilog HDL Example of an Adder/Subtractor

```

module addsub (a, b, addnsb, result);
    input [7:0] a;
    input [7:0] b;
    input addnsb;
    output [8:0] result;

    reg [8:0] result;

    always @ (a or b or addnsb)
    begin
        if (addnsb)
            result = a + b;
        else
            result = a - b;
    end
endmodule

```

VHDL Example of an Adder/Subtractor

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

PACKAGE my_package IS
    CONSTANT ADDER_WIDTH: integer := 5;
    CONSTANT RESULT_WIDTH: integer := 6;

    SUBTYPE ADDER_VALUE IS integer RANGE 0 TO 2 ** ADDER_WIDTH - 1;
    SUBTYPE RESULT_VALUE IS integer RANGE 0 TO 2 ** RESULT_WIDTH - 1;
END my_package;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.my_package.ALL;

ENTITY addsub IS
    PORT (
        a: IN ADDER_VALUE;
        b: IN ADDER_VALUE;
        addnsub: IN STD_LOGIC;
        result: OUT RESULT_VALUE
    );
END addsub;

ARCHITECTURE rtl OF addsub IS
BEGIN
    PROCESS (a, b, addnsub)
    BEGIN
        IF (addnsub = '1') THEN
            result <= a + b;
        ELSE
            result <= a - b;
        END IF;
    END PROCESS;
END rtl;

```

Infering Multipliers from HDL Code

To infer multiplier functions, synthesis tools look for multipliers and convert them to `lpm_mult` megafunctions. For devices with DSP blocks, the software may implement the `lpm_mult` function in a DSP block instead of logic, depending on device utilization. The Quartus II Fitter may also place input and output registers in DSP blocks (i.e., perform register packing) to improve performance and area utilization.



For more information on the DSP block and which functions it can implement, see the appropriate Altera device family data sheet and the *DSP Solution Center* on the Altera web site.

The following four code samples show Verilog HDL and VHDL examples for unsigned and signed multipliers that synthesis tools infer as an `lpm_mult` megafunction. Each example fits into one DSP block 9-bit element (using no extra logic cells for registers when register packing occurs).



The signed declaration in Verilog HDL is a feature of the Verilog-2001 Standard.

Verilog HDL Example of an 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
```

Verilog HDL Example of an Signed Multiplier with Input & 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
```

*VHDL Example of an Unsigned Multiplier with Input & Output Registers
(Pipelining = 2)*

```

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);
        clk: IN STD_LOGIC;
        aclr: IN STD_LOGIC;
        result: OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
    );
END unsigned_mult;

ARCHITECTURE rtl OF unsigned_mult IS
    SIGNAL a_reg, b_reg: STD_LOGIC_VECTOR (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 <= UNSIGNED(a_reg) * UNSIGNED(b_reg);
        END IF;
    END PROCESS;
END rtl;

```

VHDL Example of a Signed Multiplier

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;

ENTITY signed_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)
    );
END signed_mult;

```

```

ARCHITECTURE rtl OF signed_mult IS
    SIGNAL a_int, b_int: SIGNED (7 downto 0);
    SIGNAL pdt_int: SIGNED (15 downto 0);
BEGIN
    a_int <= SIGNED (a);
    b_int <= SIGNED (b);
    pdt_int <= a_int * b_int;
    result <= STD_LOGIC_VECTOR(pdt_int);
END rtl;

```

Inferring Multiply-Accumulators & Multiply-Adders from HDL Code

Synthesis tools detect multiply-accumulators or multiply-adders and convert them to `altsimult_accum` or `altsimult_add` megafunctions, respectively. The software then places these functions in DSP blocks.



Synthesis software only infers multiply-accumulator and multiply-adder functions if the Altera device family has dedicated DSP blocks.

A multiply-accumulator consists of a multiply operator feeding an addition operator. The addition operator feeds a set of registers that then feed the second input to the addition operator. A multiply-adder consists of two- to four-multiply operators feeding one or two levels of addition, subtraction, or addition/subtraction operators. The second-level operator, if used, is always addition. In addition to the multiply-accumulator and multiply-adder, the Quartus II Fitter can also place input and output registers into the DSP block (i.e., perform register packing) to improve performance and area utilization.

The following code samples show Verilog HDL and VHDL examples of inference for specific multiply-accumulators and multiply-adders.

Verilog HDL Example of an Unsigned Multiply-Accumulator with Input, Output & Pipeline Registers (Latency = 3)

```

module unsig_altsimult_accum (dataout, dataaa, datab, clk, aclr, clken);
    input [7:0] dataaa;
    input [7:0] datab;
    input clk;
    input aclr;
    input clken;
    output [31:0] dataout;

    reg [31:0] dataout;
    reg [7:0] dataaa_reg;
    reg [7:0] datab_reg;
    reg [15:0] multa_reg;

```

```
wire [15:0] multa;
wire [31:0] adder_out;

assign multa = dataaa_reg * datab_reg;
assign adder_out = multa_reg + dataout;

always @ (posedge clk or posedge aclr)
begin
    if (aclr)
        begin
            dataaa_reg <= 0;
            datab_reg <= 0;
            multa_reg <= 0;
            dataout <= 0;
        end
    else if (clken)
        begin
            dataaa_reg <= dataaa;
            datab_reg <= datab;
            multa_reg <= multa;
            dataout <= adder_out;
        end
    end
endmodule
```

Verilog HDL Example of a Signed Multiply-Adder (Latency = 0)

```
module sig_almult_add (dataaa, datab, dataac, datad, result);
    input SIGNED [15:0] dataaa;
    input SIGNED [15:0] datab;
    input SIGNED [15:0] dataac;
    input SIGNED [15:0] datad;
    output [32:0] result;

    wire SIGNED [31:0] mult0_result;
    wire SIGNED [31:0] mult1_result;

    assign mult0_result = dataaa * datab;
    assign mult1_result = dataac * datad;
    assign result = (mult0_result + mult1_result);
endmodule
```

*VHDL Unsigned Multiply-Adder with Input, Output & Pipeline Registers
(Latency = 3)*

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY unsignedmult_add IS
    PORT (
        a: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        c: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        d: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        clk: IN STD_LOGIC;
        aclr: IN STD_LOGIC;
        result: OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
    );
END unsignedmult_add;

ARCHITECTURE rtl OF unsignedmult_add IS
    SIGNAL a_int, b_int, c_int, d_int: STD_LOGIC_VECTOR (7 DOWNTO 0);
    SIGNAL pdt_int, pdt2_int: UNSIGNED (15 DOWNTO 0);
    SIGNAL result_int: UNSIGNED (15 DOWNTO 0);
BEGIN
    PROCESS (clk, aclr)
    BEGIN
        IF (aclr = '1') THEN
            a_int <= (OTHERS => '0');
            b_int <= (OTHERS => '0');
            c_int <= (OTHERS => '0');
            d_int <= (OTHERS => '0');
            pdt_int <= (OTHERS => '0');
            pdt2_int <= (OTHERS => '0');
            result_int <= (OTHERS => '0');

        ELSIF (clk'event AND clk = '1') THEN
            a_int <= a;
            b_int <= b;
            c_int <= c;
            d_int <= d;
            pdt_int <= UNSIGNED(a_int) * UNSIGNED(b_int);
            pdt2_int <= UNSIGNED(c_int) * UNSIGNED(d_int);
            result_int <= pdt_int + pdt2_int;
        END IF;
    END PROCESS;

    result <= STD_LOGIC_VECTOR(result_int);
END rtl;

```

VHDL Example of a Signed Multiply-Accumulator with Input, Output & Pipeline Registers (Latency = 3)

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;

ENTITY sig_altmult_accum IS
    PORT (
        a: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        clk: IN STD_LOGIC;
        accum_out: OUT STD_LOGIC_VECTOR (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 <= SIGNED (a);
            b_reg <= SIGNED (b);

            pdt_reg <= a_reg * b_reg;
            adder_out <= adder_out + pdt_reg ;
        END IF;
    END process;

    accum_out <= std_logic_vector (adder_out);
END rtl;

```

Inferring RAM from HDL Code

To infer RAM functions, synthesis tools detect sets of registers and logic that can be replaced with the `altsyncram` or `lpm_ram_dp` megafunctions, depending on the targeted device family.



Synthesis software only recognizes RAM blocks for device families that have dedicated RAM blocks.

Synthesis tools recognize single port and simple dual-port (one read and one write port) RAM blocks. The software may not infer very small RAM blocks because very small RAM blocks can typically be implemented more efficiently by using the registers in regular logic.



If your design contains a RAM block that your synthesis tool does not recognize and infer, it may use a large amount of memory and could potentially cause runtime compilation problems.



For certain RAM configurations in certain device families, using a RAM megafunction may slightly change the design functionality if the RAM reads from and writes to the same location. In this scenario, the software generally issues a warning. If you are using Quartus II integrated synthesis, Quartus II Help explains the condition under which the functionality changes.

When using a formal verification flow, Altera recommends that you create RAM blocks in separate entities or modules containing 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 automatically black-boxed because formal verification tools do not support RAM megafunctions. The software issues a warning message when this occurs. If the entity or module contains any additional logic outside the RAM, this logic is also black-boxed and cannot be verified.

The following code samples show Verilog HDL and VHDL examples that infer single- and dual-clock synchronous RAM. Depending on the device family's dedicated RAM architecture, the RAM may need to be synchronous.



Refer to the appropriate Altera device family data sheet or handbook for more information about your specific device at www.altera.com/literature.

For the dual-clock examples—if you are reading and writing to the same address—the functionality of the inferred megafunction may differ from the original HDL code. (Synthesis tools issues a warning to inform you of this functional difference.)

Verilog HDL Example of a Single-Clock Synchronous RAM

```
module ram_infer (q, a, d, we, clk);
    output [7:0] q;
    input [7:0] d;
    input [6:0] a;
    input we, clk;

    reg [6:0] read_add;
    reg [7:0] mem [127:0];
```

```

always @ (posedge clk) begin
    if (we)
        mem[a] <= d;
    read_add <= a;
end

assign q = mem[read_add];
endmodule

```

Verilog HDL Example of a Dual-Clock Synchronous RAM

```

module ram_dual (q, addr_in, addr_out, d, we, clk1, clk2);
    output [7:0] q;
    input [7:0] d;
    input [6:0] addr_in;
    input [6:0] addr_out;
    input we, clk1, clk2;

    reg [6:0] addr_out_reg;
    reg [7:0] q;
    reg [7:0] mem [127:0];

    always @ (posedge clk1)
begin
    if (we)
        mem[addr_in] <= d;

    end
    always @ (posedge clk2) begin
        q <= mem[addr_out_reg];
        addr_out_reg <= addr_out;
    end
endmodule

```

VHDL Example of a Single-Clock Synchronous RAM

```

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;
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;

```

VHDL Example of a Dual-Clock Synchronous RAM

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY ram_dual 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 ram_dual;

ARCHITECTURE rtl OF ram_dual 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;

```

The following code samples show Verilog HDL and VHDL code examples of RAM with asynchronous read addresses and registered outputs.

The implementation of RAM example code in the following samples varies depending on the dedicated RAM architecture of the appropriate device family. For example, implementing asynchronous read addresses in an APEX device's RAM block is straightforward because the APEX architecture supports asynchronous read addresses. However, read addresses in Stratix® devices and most newer device families must be registered; therefore, you cannot directly implement the asynchronous RAM example code in the following samples. To implement the asynchronous RAM example from the Stratix architecture by inferring an `altsyncram` megafunction, synthesis tools may move the output registers to the inputs of the RAM block. If the read and write clocks are not the same, moving the output registers to the inputs of the RAM block may slightly change the functionality. In these circumstances, the software issues a warning. When using Quartus II integrated synthesis, Quartus II Help explains the differences.

Verilog HDL Example of a Single-Clock Synchronous RAM with Asynchronous Read Address

```

module ram (clock, data, write_address, read_address, we, q);
parameter ADDRESS_WIDTH = 4;
parameter DATA_WIDTH      = 8;

input clock;
input [DATA_WIDTH-1:0] data;
input [ADDRESS_WIDTH-1:0] write_address;
input [ADDRESS_WIDTH-1:0] read_address;
input we;
output [DATA_WIDTH-1:0] q;

reg [DATA_WIDTH-1:0] q;
reg [DATA_WIDTH-1:0] ram_block [2**ADDRESS_WIDTH-1:0];

always @ (posedge clock)
begin
  if (we)
    ram_block[write_address] <= data;
end

```

```

        q <= ram_block[read_address];
      end
endmodule

```

VHDL Example of a Single-Clock Synchronous RAM with Asynchronous Read Address

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY ram IS
  GENERIC (
    ADDRESS_WIDTH: integer := 4;
    DATA_WIDTH: integer := 8
  );
  PORT (
    clock: IN std_logic;
    data: IN STD_LOGIC_VECTOR(DATA_WIDTH - 1 DOWNTO 0);
    write_address IN STD_LOGIC_VECTOR (ADDRESS_WIDTH - 1 DOWNTO 0);
    read_address IN STD_LOGIC_VECTOR (ADDRESS_WIDTH - 1 DOWNTO 0);
    we: IN STD_LOGIC;
    q: OUT STD_LOGIC_VECTOR (DATA_WIDTH - 1 DOWNTO 0)
  );
END ram;

ARCHITECTURE rtl OF ram IS
  TYPE RAM IS ARRAY(0 TO 2 ** ADDRESS_WIDTH - 1) OF
  std_logic_vector(DATA_WIDTH - 1 DOWNTO 0);
  SIGNAL ram_block: RAM;
BEGIN
  PROCESS (clock)
  BEGIN
    IF (clock'event AND clock = '1') THEN
      IF (we = '1') THEN
        ram_block(TO_INTEGER(UNSIGNED(write_address))) <= data;
      END IF;
      q <= ram_block(TO_INTEGER(UNSIGNED(read_address)));
    END IF;
  END PROCESS;
END rtl;

```

Inferring ROM 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.



Synthesis software only recognizes ROM functions for device families that have dedicated memory blocks.

ROMs are inferred when you have a case statement where a value is being 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 has to meet a minimum size requirement to be inferred and placed into memory.



Because formal verification tools do not support ROM megafunctions, Quartus II integrated synthesis does not infer ROM megafunctions when a third-party formal verification tool is selected under **EDA Tool Settings** in the **Settings** dialog box (Assignments menu).

When using a formal verification flow, Altera recommends that you create ROM blocks in separate entities or modules containing only the ROM blocks logic because you may need to treat the entity and module as a black box during formal verification.

The following code samples show Verilog HDL and VHDL examples that infer synchronous ROM. Depending on the device family's dedicated RAM architecture, the ROM may need to be synchronous; consult the device family data sheet for details. For device architectures with synchronous RAM blocks, such as Stratix devices and most 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 Stratix 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 software generally issues a warning. When using Quartus II integrated synthesis, Quartus II Help explains the condition under which the functionality changes.

Verilog Example of an 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
```

VHDL Example of a 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;
```

Inferring Shift Registers 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

 Because formal verification tools do not support shift register megafunctions, Quartus II integrated synthesis does not infer `altshift_taps` megafunctions when a third-party formal verification tool is selected under **EDA Tool Settings** in the **Settings** dialog box (Assignments menu).

When 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.

Synthesis software recognizes shift registers only for device families that have dedicated RAM blocks and use certain guidelines to determine the best implementation. The following guidelines are followed in Quartus II integrated synthesis and are generally followed by third-party EDA tools as well:

- 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 `altshift_taps` megafunctions 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 Stratix II, Stratix, Cyclone™ II, and Cyclone devices, the software determines whether to infer `altshift_taps` megafunctions 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 ($W \times N \times L \geq 32$).



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 different sizes of shift registers, external decode logic (using LEs or asynchronous line multiplexer (ALMs)) are required to implement the function, which eliminates the advantage 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 simulation tools because their node names do not exist after synthesis.

The following code sample shows a Verilog HDL example of a simple, single-bit wide, 64-bit long shift register. The 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.

Verilog HDL Example of a 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
```

The following code samples shows a Verilog HDL and an VHDL example of an 8-bit wide, 64-bit long shift register ($W > 1$ and $M = 64$) with evenly spaced taps at 15, 31, and 47. The software implements this function in a single `altshift_taps` megafunction and maps it to RAM in supported devices.

Verilog HDL Example of an 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

```

VHDL Example of an 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;

```

Device-Specific Coding Recommendations

This section provides device-specific coding recommendations for Altera device architectures. Designing specific logic structures to match the appropriate Altera device architecture can provide significant improvements in quality of results.

Secondary Control Signals in Registers or Flipflops

FPGA device architectures are based on registers, or 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 your device family data sheet or handbook 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 it is always possible to get functionally correct results. However, if your design requirements are flexible in terms of which control signals are used and in what priority, you can achieve the most efficient results by matching the device architecture. If the priority of the signals in your design is not the same as the target architecture, then extra logic may be required to implement the control signals.



The priority order for secondary control signals in Altera devices may be different than the order for other vendors' devices, so if your design requirements are flexible in this area, it is a good idea to check your secondary control signals when migrating designs between FPGA vendors.

The signal order is the same for all Altera device families, although as noted above, not all device families provide every signal. The priority order is shown here:

1. Asynchronous Clear, `aclr`
2. Preset
3. Asynchronous Load, `aload`
4. Enable, `ena`
5. Synchronous Clear, `sclr`
6. Synchronous Load, `sload`
7. Data In

The examples below provide Verilog HDL and VHDL code that create a register with the `aclr`, `aload`, and `ena` control signals listed above.

The preset signal is not available on recent device families, because it was replaced with the more flexible `aload` signal, so it is not included in the examples. Creating many registers with different `sload` and `sclr` signals can make it difficult for the Quartus II Fitter to pack the registers into logic array blocks (LABs), since the `sclr` and `sload` signals are LAB-wide signals. Therefore, synthesis tools typically restrict their use to certain examples such as arithmetic chains (e.g., counters) or wide multiplexers where there are enough registers with common signals to allow good LAB packing. If you do use these additional control signals, use them in the priority order that matches the device architecture. To ensure that you can achieve the most efficient results, the `sclr` signal should have a higher priority than the `sload` signal in the same way that `aclr` has higher priority than `aload` in the following examples.



dff_all.v does not have `adata` on the sensitivity list, but **dff_all.vhd** does. This is a limitation of the Verilog HDL language —there is no way to describe an asynchronous load signal (where `q` toggles if `adata` toggles while `aload` is high). All synthesis tools should infer an `aload` signal from this construct despite this limitation, although you may see information or warning messages from the synthesis tool.

Verilog Example of an HDL D-Flipflop (Register) with Control Signals

```
moduledff_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
```

VHDL Example of a D-Flipflop (Register) with 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;
```

Tri-State Signals

When targeting Altera devices, you should only use tri-state signals 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 multiplexing logic, but Altera does not recommend this coding practice.



In hierarchical, block-based, or incremental design flows, a hierarchical boundary can not contain any bidirectional ports.

The following code samples are simple examples for creating tri-state bidirectional signals.

Verilog HDL Example of a Tri-State Signal

```
module tristate (myinput, myenable, mybidir);
    input myinput, myenable;
    inout mybidir;
    assign mybidir = (myenable ? myinput : 1'bZ);
endmodule
```

VHDL Example of a 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 that uses a large adder

tree is a finite impulse response (FIR) correlator; using a pipelined binary or ternary adder tree appropriately can greatly improve your quality of results.

This section explains why coding recommendations are different for Altera four-input look-up table (LUT) devices (e.g., Stratix, APEX 20K, and FLEX 10K devices) and the six-input LUT logic structures available in Stratix II devices.

Architectures With Four-Input LUTs in Logic Elements

Architectures such as Stratix, APEX 20K, and FLEX 10K devices contain four-input LUTs as the standard combinational structure in the logic element (LE).

If your design can tolerate pipelining, the fastest way to add three numbers A , B , and C , in Stratix, APEX 20K, or FLEX 10K devices is to add $A + B$, register the output, and then add the registered output to C . Adding $A + B$ takes one level of logic (i.e., 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 example that follows, five numbers A , B , C , D , and E are added. Adding five numbers in Stratix, APEX 20K, or FLEX 10K devices requires four adders and three levels of registers for a total of 64 LEs (for 16-bit numbers).

Verilog HDL Example of a 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
endmodule
```

```
// 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 Six-Input LUTs in Adaptive Logic Modules

Because the Stratix II architecture uses a six-input LUT in its basic logic structure, the adaptive logic module (ALM), Stratix II devices benefit from a different coding style. Specifically, Stratix II device ALMs can simultaneously add three bits. Thus, the tree in the previous example needs to be two levels deep and contain just two add-by-three inputs instead of four add-by-two inputs.

Again, although the code in the previous example successfully compiles for Stratix II devices, it is not efficient and does not take advantage of the six-input Adaptive LUT (ALUT). By restructuring the tree as a ternary tree the design becomes much more efficient, significantly improving density utilization. Therefore, when targeting Stratix II devices, large pipelined binary adder trees designed for four-input LUT architectures should be rewritten to take advantage of the Stratix II device architecture.

The following example uses just 32 ALUTs in a Stratix II device—more than a four-to-one advantage over the number of LUTs in the prior example implemented in a Stratix device.



You cannot pack a Stratix II 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 ALM.

Verilog HDL Example of a 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
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 non-pipelined 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 $\text{sum} = (\text{A} + \text{B} + \text{C}) + (\text{D} + \text{E})$ is more likely to create the optimal implementation of a 3-input adder for $\text{A} + \text{B} + \text{C}$ followed by a 3-input adder for $\text{sum1} + \text{D} + \text{E}$ than the code without the parenthesis. If you don't add the parenthesis, the synthesis tool may partition the addition in a way that is not optimal for the architecture.

General Coding Recommendations

This section provides general coding recommendations, specifically regarding latches, state machines, and multiplexers.

Latches

When 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 latches being inferred.



Latches have a limited support in formal verification tools. Therefore, it is especially important to ensure that you do not infer latches unintentionally, e.g. through an incomplete CASE statement, when using formal verification.

The `full_case` attribute can be used in Verilog HDL designs to indicate that non-specified cases can be treated as “don’t care.” However, using the `full_case` attribute may lead to simulation mismatches because it is a synthesis-only attribute.



See 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” assignments on the default conditions tend to prevent latch generation. Synthesis software

generally treats unknowns as “don’t care” conditions to optimize logic. For the best logic optimization, assign the default CASE or final ELSE value to “don’t care” instead of a logic value.

The following shows example VHDL code that prevents an unintentional latch. Without the final ELSE clause, the code creates unintentional latches to cover the remaining combinations of the sel inputs. When targeting a Stratix device with the following code, omitting the final ELSE condition may cause the synthesis software to use up to six LEs instead of the three it uses with the ELSE statement. Also, assigning the final ELSE value to 1 instead of X may result in slightly more LEs.

VHDL Example of 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;
```

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 using state machines.

To achieve the best results on average, synthesis tools often use one-hot encoding for FPGA devices and minimal-bits encoding for CPLD devices, although the choice of implementation may vary for different state machines. See your synthesis tool’s documentation for tool-specific ways to control how state machines are encoded.



For information about state machine encoding in Quartus II integrated synthesis, refer to the State Machine Processing section in *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 performance, 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 to avoid generation of unwanted latches during synthesis.
- Assign a default clause to direct the state machine in case it accidentally reaches an unused state.
- 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 make the output logic of the state machine 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 an asynchronous reset and an asynchronous load at the same time, the Quartus II software, for example, generates regular logic rather than inferring a state machine.



See the following sections for additional guidelines and coding examples using “[Verilog HDL State Machines](#)” on page 5–33 and “[VHDL State Machines](#)” on page 5–36.

Verilog HDL State Machines

To ensure proper recognition and inference of Verilog HDL state machines, observe the following additional Verilog-specific guidelines. The enforcement of some of these guidelines may be specific to the Quartus II integrated synthesis tool. See your synthesis tool’s document for more tool-specific coding recommendations.

- Represent the status in a state machine with the `parameter` data types and use the parameters to make state assignments. This implementation makes the state machine easier to read and reduces the risks of errors during coding.



Altera recommends against the direct use of integer values for state variables such as `next_state <= 0`. However, integer use 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 such as 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 used to create an output as follows:

```

output out1
case (state)
  state_0: begin
    if (ena) out1 <= state_1;
    else out1 <= state_2;
    next_state <= state_2;
  end
  state_1: begin
    ...
  endcase
    
```

Verilog HDL State Machine Coding Example

The module `verilog_fsm` that follows is an example of a typical Verilog HDL state machine implementation.

This machine has five states. The asynchronous reset sets the variables `state` to `state_0`. The sum of `in_1` and `in_2` is used as an output of the state machine in `state_1` and `state_2`. The difference of `in_1` and `in_2` is used in the state `state_1` and `state_3`. The temporary variables `tmp_out_0` and `tmp_out_1` are used to store the sum and the difference of `in_1` and `in_2`. The use of these temporary variables in the various states of the state machine ensures proper resource sharing between these mutually exclusive states.

Verilog HDL Example of a 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 could 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 it conserves the state names throughout synthesis.

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 risks of errors during coding. If the state is not represented by an enumerated type, the Quartus II synthesis software, for example, does not recognize the state machine. Instead, it is implemented as regular logic gates and registers, and it is not listed as a state machine in the Analysis & Synthesis report.

VHDL State Machine Coding Example

The following entity `vhd1_fsm` is an example of a typical VHDL state machine implementation.

This state machine has five states. The asynchronous reset sets the variable state to `state_0`. The sum of `in1` and `in2` is used as 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` are used to store the sum and the difference of `in1` and `in2`. The use of these temporary variables in the various states of the state machine ensures the proper resource sharing between these mutually exclusive states.

VHDL Example of a 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
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;
        END CASE;
    END PROCESS;
END;

```

```

    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 multiplexing logic, you can ensure the most efficient implementation in your Altera device. This section discusses some common pitfalls and provides design guidelines to achieve optimal resource utilization for multiplexer designs. The section also describes the different types of multiplexers, and how they are implemented in the 4-input (LUT) found in many FPGA architectures, such as Altera's Stratix devices.



Devices with 6-input LUTs (Stratix II devices) are not specifically discussed here. Many of the principles and techniques for optimization are similar, but the device utilization is different in these devices. Devices with 6-input LUTs can implement wider multiplexers in one ALM than can be implemented in the 4-input LUT of an LE.

Types of Multiplexers

This first sub-section discusses how multiplexers or “muxes,” are created from various types of HDL code. CASE statements, IF statements, and state machines are all common sources of multiplexing logic in designs. These HDL structures create different types of multiplexers including binary multiplexers, selector multiplexers, and priority multiplexers. Understanding how multiplexers arise 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. The “[Verilog HDL Example of a Simple Binary-Encoded “Case” Statement](#)” example below shows Verilog HDL code that describes a simple 4:1 binary multiplexer.

Verilog HDL Example of a Simple Binary-Encoded “Case” Statement

```
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 muxes can be constructed using the 4:1 mux; constructing an N -input multiplexer ($N:1$ mux) from a tree of 4:1 muxes 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 mux are essentially one-hot encoded. [“Verilog HDL Example of a Simple One-Hot-Encoded “Case” Statement”](#) example below shows a simple Verilog HDL code samples that describes a one-hot selector multiplexer.

Verilog HDL Example of a Simple 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 = "X";
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 using two AND gates and an OR gate. The outputs of these LUTs can be combined using 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

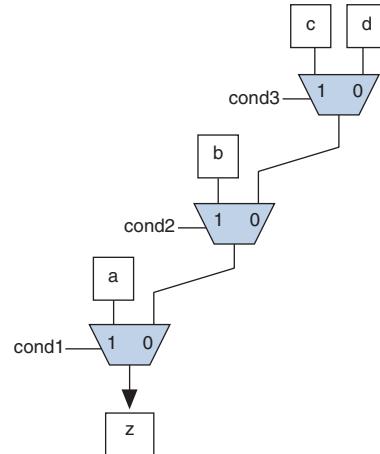
In priority multiplexers, the select logic implies a priority, so the options to select the correct item must be checked in order. These structures commonly arise from IF, ELSE, WHEN, SELECT, or ? : statements in VHDL or Verilog HDL. The example VHDL code in the [“VHDL Example of an IF Statement Implying Priority”](#) example below is likely to result in the implementation illustrated schematically in [Figure 5–1](#).

VHDL Example of an IF Statement Implying Priority

```
IF cond1 THEN z <= a;
ELSIF cond2 THEN z <= b;
ELSIF cond3 THEN z <= c;
ELSE z <= d;
END IF;
```

Notice that the multiplexers shown in Figure 5–1 form a chain, evaluating each condition, or select bit, one at a time.

Figure 5–1. Priority Multiplexer Implementation of an IF Statement



An N -input priority mux uses a LUT for every 2:1 multiplexers in the chain, requiring $N-1$ LUTs. In addition, this chain of multiplexers is generally bad for delay since the critical path through the logic traverses every multiplexer in the chain.

Avoid priority muxes where 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 mux instead of the priority mux. If delay through the structure is important in a multiplexing design that requires priority, consider recoding the design to reduce the number of logic levels.

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 directs the synthesis tool how to deal with these cases, and is required by the Verilog HDL and VHDL language specifications.

Some designs do not have a requirement for the outcome in the unused cases, often because it is assumed that these cases will not arise. In these situations, 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 they use different speed and area optimizations.

In general, to obtain best results, explicitly define your invalid CASE selections with a separate DEFAULT or OTHERS statement instead of combining the invalid cases with one of the defined cases.

If you do not care about the value in the invalid cases, explicitly say so by assigning the "X" logic value for these cases instead of choosing another value. This assignment should allow your synthesis tool to make the best area optimizations.

You may want to experiment with different DEFAULT or OTHERS assignments for your HDL design and your synthesis tool to test the effect they have on your logic utilization.

Implicit Defaults

The IF statements in Verilog HDL and VHDL can be a convenient way of specifying conditions that don't easily lend themselves to a CASE-type approach. However, these 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 if it is not specified. These implicit defaults can cause additional complexity in a multiplexing design.

The code sample in the "[VHDL Example of an IF Statement with Implicit Defaults](#)" example below appears to represent a 4:1 multiplexer; there are four inputs (a, b, c, d) and one output (z).

VHDL Example of an 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;

```

However, each of the three separate IF statements in the code has an implicit ELSE condition that is not specified. Since 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. The code sample in the “[VHDL Example of an IF Statement with Default Conditions Explicitly Specified](#)” example shows code with the same functionality as the code in the “[VHDL Example of an IF Statement with Implicit Defaults](#)” on page 5–42 example but specifies the ELSE cases explicitly.

VHDL Example of an IF Statement with Default Conditions Explicitly Specified

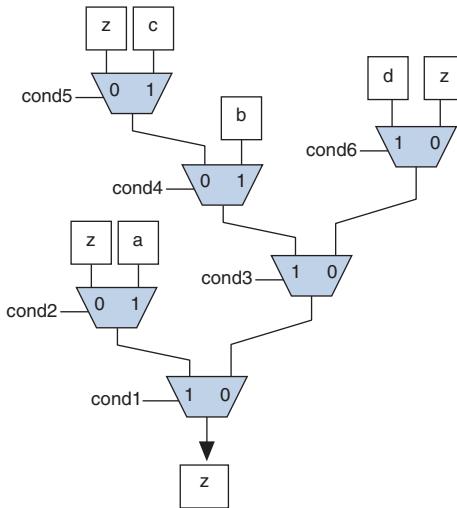
```

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 5–2](#) is a schematic representation of the code in the “[VHDL Example of an IF Statement with Default Conditions Explicitly Specified](#)” on page 5–42, illustrating that although there are only four inputs, the multiplexing logic is significantly more complicated than a basic 4:1 mux.

Figure 5–2. Multiplexer Implementation of an IF Statement with Implicit Defaults Shown



You can do several things in these cases to simplify the multiplexing logic and remove the unneeded defaults. The most optimal way may be to recode the design so it takes the structure of a 4:1 CASE statement.

Alternately, or if the 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, question 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 the logic utilization required to implement your design.

Degenerate Multiplexers

A degenerate multiplexer is one 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 muxes so that they take advantage of the efficient logic utilization possible with full binary muxes.

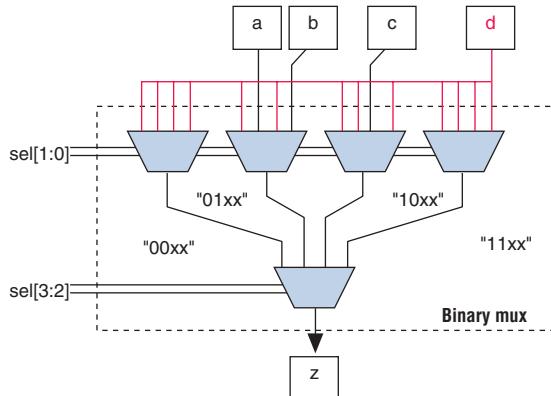
The number of select lines in a binary multiplexer normally dictates how big a mux is needed to implement the desired function. For example, the mux structure represented in [Figure 5–3](#) on page [5–44](#) has four select lines

and could implement a binary multiplexer with 16 inputs. However, the figure does not use all 16 inputs and thus is considered a “degenerate” 16:1 mux.

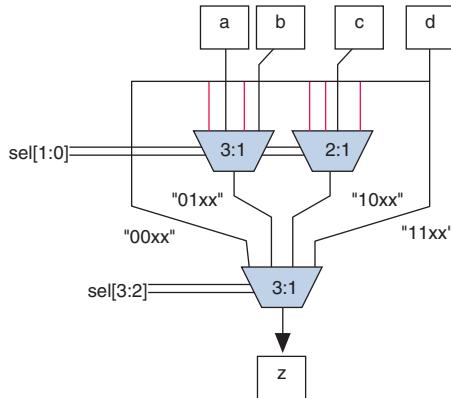
VHDL Example of a CASE Statement Describing a Degenerate Multiplexer

```
CASE sel[3:0] IS
    WHEN "0101" => z <= a;
    WHEN "0111" => z <= b;
    WHEN "1010" => z <= c;
    WHEN OTHERS => z <= d;
END CASE;
```

Figure 5–3. Binary Multiplexer Implementation of a Degenerate Multiplexer



In the example in [Figure 5–3](#), the first and fourth muxes in the top level can easily be eliminated since all four inputs to each mux are the same value, and the number of inputs to the other multiplexers can be reduced, as shown in [Figure 5–4](#).

Figure 5-4. Optimized Version of the Degenerate Binary Multiplexer

Implementing this version of the multiplexer still requires at least 5 4-input LUTs, two for each of the remaining 3:1 muxes and one for the 2:1 mux. This design selects an output from only four inputs, a 4:1 binary mux can be implemented optimally in 2 LUTs, so this degenerate multiplexer tree is reducing the efficiency of the logic.

You can improve the logic utilization of this type of structure by recoding the select lines to implement a full 4:1 binary mux. “[VHDL Example of a Recoder Design for Degenerate Binary Multiplexer](#)” below provides code for a recoder design that translates the original select lines into a signal `z_sel` with binary encoding, and “[VHDL Example of a 4:1 Binary Multiplexer Design](#)” below provides code to implement the full binary mux.

VHDL Example of a Recoder Design for Degenerate Binary Multiplexer

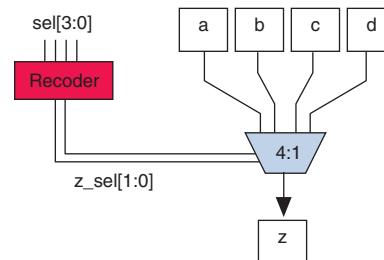
```
CASE sel[3:0] IS
  WHEN "0101" => z_sel <= "00";
  WHEN "0111" => z_sel <= "01";
  WHEN "1010" => z_sel <= "10";
  WHEN OTHERS => z_sel <= "11";
END CASE;
```

VHDL Example of a 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 to control the 4:1 binary multiplexer that chooses between the four inputs `a`, `b`, `c`, and `d`, as illustrated in [Figure 5–5](#). The complexity of the select lines is handled in the recoder, and the data multiplexing is performed with simple binary select lines enabling the most efficient implementation.

Figure 5–5. Binary Multiplexer with Recoder



The recoder design can be implemented in two LUTs and the efficient 4:1 binary mux uses two LUTs, for a total of four LUTs. The original degenerate mux 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 do the encoding, more logic may be saved performing the data multiplexing.

Buses of Multiplexers

The inputs to multiplexers are often buses of data inputs where the same multiplexing function is performed on a set of data inputs in the form of 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 mux buses.

For example, the recoding technique discussed in the previous section can often be used in buses that involve multiplexing. Recoding the select lines may only need to be done 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 muxes.

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 recoded design uses only two LUTs, and the select lines only need to be recoded once for the entire bus. The binary 4:1 mux requires two LEs per bit of the bus. The

total logic utilization for the recoded version could be $2 + (2 \times 32)$ or 66 LUTs for the whole bus, as compared to 160 LUTs for the original version! The savings in logic become much more obvious when the mux works across wide buses.

Using techniques to optimize degenerate muxes, removing unneeded implicit defaults, and choosing the optimal DEFAULT or OTHERS case can play an important role when optimizing buses of multiplexers.

Quartus II Option for Multiplexers Restructuring

Quartus II integrated synthesis provides the **Restructure Multiplexers** logic option that can help extract and optimize buses of muxes during synthesis. In certain situations, this option performs some of the recoding functions described above automatically without actually changing your HDL code.



For details, refer to the Restructure Multiplexers subsection in the *Quartus II Integrated Synthesis* chapter in Volume 1 of the *Quartus II Handbook*.

Conclusion

Keep the targeted device architecture in mind when selecting your coding style, as certain coding styles can dramatically improve performance results. To improve your design's performance and area utilization, take advantage of advanced device features such as memory and DSP blocks, as well as the specific architecture of the targeted Altera device, and follow the coding recommendations presented in this chapter.



For additional optimization recommendations, see the *Design Optimization for Altera Devices* chapter in Volume 2 of the *Quartus II Handbook*.

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 & 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 4.2 software.

This section includes the following chapters:

- [Chapter 6, Quartus II Integrated Synthesis](#)
- [Chapter 7, Synplicity Synplify & Synplify Pro Support](#)
- [Chapter 8, Mentor Graphics LeonardoSpectrum Support](#)
- [Chapter 9, Mentor Graphics Precision RTL Synthesis Support](#)
- [Chapter 10, Synopsys FPGA Compiler II BLIS & Quartus II LogicLock Design Flow](#)
- [Chapter 11, Synopsys Design Compiler FPGA Support](#)
- [Chapter 12, Analyzing Designs with the Quartus II RTL Viewer & Technology Map Viewer](#)

Revision History

The table below shows the revision history for [Chapters 6 to 12](#).

Chapter(s)	Date / Version	Changes Made
7	Dec. 2004 v3.0	<ul style="list-style-type: none"> ● Chapter 7 was formerly Chapter 8 in version 4.1. ● Added documentation of incremental synthesis feature ● New functionality for Quartus II software 4.2
	June 2004 v2.0	<ul style="list-style-type: none"> ● Updates to tables, figures. ● New functionality for Quartus II software 4.1.
	Feb. 2004 v1.0	Initial release.
8	Dec. 2004 v2.1	<ul style="list-style-type: none"> ● Chapter 8 was formerly Chapter 9 in version 4.1. ● Updated information. ● New functionality for Quartus II software 4.2. ● Updated figure 8-1.
	June 2004 v2.0	<ul style="list-style-type: none"> ● Updates to tables, figures. ● New functionality for Quartus II software 4.1.
	Feb. 2004 v1.0	Initial release.
9	Dec. 2004 v2.1	<ul style="list-style-type: none"> ● Chapter 9 was formerly Chapter 10 in version 4.1. ● Updates to tables and figures. ● New functionality for Quartus II software 4.2.
	June 2004 v2.0	<ul style="list-style-type: none"> ● Updates to tables and figures. ● New functionality for Quartus II software 4.1.
	Feb. 2004 v1.0	Initial release.
10	Dec. 2004 v2.1	<ul style="list-style-type: none"> ● Chapter 10 was formerly Chapter 11 in version 4.1. ● Updated information. ● New functionality in Quartus II software 4.2. ● Updated tables and figures.
	June 2004 v2.0	<ul style="list-style-type: none"> ● Updates to tables, and figures. ● New functionality for Quartus II software 4.1.
	Feb. 2004 v1.0	Initial release.
11	Dec. 2004 v1.0	<ul style="list-style-type: none"> ● Chapter 11 was formerly Chapter 12 in version 4.1. ● Updates to tables, figures. ● New functionality for Quartus II software 4.2.
	June 2004 v1.0	No change to document content.
	Feb. 2004 v1.0	Initial release.

Chapter(s)	Date / Version	Changes Made
12	Dec. 2004 v1.1	<ul style="list-style-type: none">● Chapter 12 was formerly Chapter 13 in version 4.1.● Updated information.● New functionality for Quartus II software 4.2.● Moved figure 12-3 within the chapter.
	June 2004 v1.0	Initial release.
13	Dec. 2004 v2.1	<ul style="list-style-type: none">● Chapter 13 was formerly Chapter 14 in version 4.1.● Updates to tables and figures.● New functionality for Quartus II software 4.2.
	June 2004 v 2.0	<ul style="list-style-type: none">● Updates to tables, and figures.● New functionality for Quartus II software 4.1.
	Feb. 2004 v1.0	Initial release.

qii51008-3.0

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 the Verilog and VHDL hardware description languages (HDLs), 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, standalone solution for system-on-a-programmable-chip (SOPC) designs.

This chapter documents the language support in the Quartus II software, and explains how to improve and control your Quartus II synthesis results by using incremental synthesis, Quartus II synthesis options, and controlling the inference of architecture-specific megafunctions.

Additionally, this chapter explains some of the node-naming conventions used during synthesis to help you better understand your synthesized design. Scripting techniques for applying all the options and settings described are also provided.

Language Support

This section explains the Quartus II software's integrated synthesis support for both HDL and schematic design entry. All supported languages, as well as netlists generated by third-party synthesis tools, can be mixed in a single Quartus II project.

Verilog HDL

The Quartus II Compiler's analysis and synthesis module supports the Verilog-1995 standard (IEEE Std. 1364-1995) and the Verilog-2001 standard (IEEE Std. 1364-2001) constructs. You can select which standard to use in the **Verilog version** section of the **Verilog HDL Input** page under **Analysis & Synthesis Settings** in the **Settings** dialog box (Assignments menu). The Quartus II Compiler uses the Verilog-2001 standard by default.



The Verilog HDL code samples provided in this document follow the Verilog-2001 standard unless otherwise specified.

Supported Verilog-2001 standard constructs include:

- Generate statements: generate and genvar
- localparam constants
- Pre-processor statements such as `elsif, `line, `ifdef, `file, and `default_nettype
- Signed declarations for all variables
- Operators such as **, <<<, and >>>
- Attributes using the syntax (* name = value *)
- Indexed part selects using +: and -:
- Combinational logic sensitivity wild card token @*
- Combined port and data type declarations
- ANSI-style port lists
- In-line parameter passing by name (explicit redefinition using #)
- Multi-dimensional arrays

The Quartus II software does not support Verilog-2001 libraries and configurations.



The Quartus II software supports case-sensitive Verilog HDL code, in accordance with the Verilog HDL standard. For more information on specific syntax features and language constructs, refer to “Quartus II Verilog HDL Support” in Quartus II Help.

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 or current file location). When searching for a relative path, the Quartus II software first searches relative to the project directory. If the software cannot find the file, it searches relative to the directory location of the file.

VHDL

The Quartus II Compiler's analysis and synthesis module supports the VHDL 1987 (IEEE Std. 1076-1987) and VHDL 1993 (IEEE Std. 1076-1993) standards. You can select which standard to use in the **VHDL version** section of the **VHDL Input** page under **Analysis & Synthesis Settings** in the **Settings** dialog box (Assignments menu). The Quartus II Compiler uses the VHDL 1993 standard by default.



The VHDL code samples provided in this document follow the VHDL 1993 standard.



For information on specific syntax features and language constructs, refer to “Quartus II VHDL Support” in Quartus II Help.

VHDL Libraries

The Quartus II software includes the standard IEEE libraries and a number of vendor-specific VHDL libraries. You can also create custom VHDL libraries to store your own VHDL design units.



Unlike the MAX+PLUS® II software and earlier versions of the Quartus II software, Quartus II software versions 2.1 and later do not support pre-compiled libraries.

The **IEEE** library includes the standard VHDL packages `std_logic_1164`, `numeric_std`, and `numeric_bit`. The **STD** library is part of the VHDL language standard and includes 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 packages such as `maxplus2` in the **ALTERA** library, `altera_mf_components` in the **ALTERA_MF** library, and `lpm_components` in the **LPM** library



For a complete listing of library and package support, see *Using Quartus II Packages* in the Quartus II Help.

To synthesize a design that uses VHDL design files, add each file to your Quartus II project by choosing **Add/Remove Files In Project** (Project menu).

The Quartus II software requires that you analyze a design unit prior to its use. Because the Compiler analyzes files in the order in which they are listed, a file that declares a design unit must precede a file that uses the design unit. For example, you must list the file that declares a package before a file that refers to the package with a **USE** clause. To change the order of the files in the **File name** list, select a file and click **Up** or **Down**.

By default, the Quartus II software compiles all VHDL files into the **work** library. If a VHDL file refers to a library that does not exist, or if the library does not contain a referenced design unit, then the software searches the **work** library. This default behavior allows the Quartus II software to compile most VHDL designs with minimal setup.

Prior to analyzing the sources files in a design, you may specify a different destination library for the design units in a VHDL source file. You can use one of the following three methods to specify a destination library:

- In the **Settings** dialog box (Assignments menu)
- In the Quartus II Settings File (.qsf), or with a Tcl command
- In the VHDL file itself, using a synthesis directive

When the Quartus II Compiler analyzes the file (in a flow that requires analysis of the source file), it stores the analyzed design units in the file's destination library.

 A VHDL design cannot contain two or more entities with the same name, even if they are compiled into separate custom libraries.

Specifying a Destination Library Name in the Settings Dialog Box

To specify a library name for one of your VHDL files:

1. Choose the appropriate VHDL file in the **File Name** list on the **Files** page of the **Settings** dialog box (Assignments menu).
2. Click **Properties**.
3. In the **File Properties** dialog box, select **VHDL File** from the **Type** list (if it is not already selected).
4. Type the desired library name in the **Library** field.

Specifying a Destination Library Name in the QSF or Using Tcl

In the QSF or through Tcl commands, you can specify the VHDL library name with the **-library** argument associated with the **VHDL_FILE** assignment.

For example, the following QSF or Tcl assignment specifies that the Quartus II software analyze **my_file.vhd** and store its contents (design units) in the VHDL library **my_lib**.

```
set_global_assignment VHDL_FILE my_file.vhd -library my_lib
```

For more information on Tcl scripting, refer to “Scripting Support” on page 6–51.

Specifying a Destination Library Name in Your 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 (i.e., a package declaration, an entity declaration, or a configuration), using one of the supported keywords for synthesis directives, i.e. `synthesis`, `pragma`, `synopsys`, or `exemplar`.

For more information on specifying synthesis directives, refer to “[Synthesis Directives](#)” on page 6-17.

The `library` directive overrides the default library destination work, the library setting specified for the current file through the **Settings** dialog box (Assignments menu), an existing QSF setting, 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.

The following is an example using the `library` synthesis directive to create a library called `my_lib` that contains the design unit `my_entity`.

```
-- synthesis library my_lib
library ieee;
use ieee.std_logic_1164.all;
entity my_entity(...)
end entity my_entity;
```

 Specifying the library name through the the **Settings** dialog box (Assignments menu), editing the QSF, or using the Tcl interface allows you to specify only a single destination library for all the design units in a given source file. This synthesis directive allows you to change the destination VHDL library within a source file, providing the option of organizing the design units in a single file into different libraries, rather than just a single library.

The Quartus II software gives an error if you use the `library` directive within a design unit.

AHDL

The Quartus II Compiler’s analysis and synthesis module fully supports the Altera Hardware Description Language (AHDL).

AHDL designs use Text Design Files (`.tdf`). AHDL Include Files (`.inc`) can be imported into a TDF using an AHDL `include` statement. Altera provides INC files for all megafunctions shipped with the Quartus II software.



For information on specific syntax features and language constructs, see *Using AHDL in the Quartus II Software* in Quartus II Help.



The AHDL language does not support the synthesis directives or attributes described in this chapter.

Schematic Design Entry

The Quartus II Compiler's analysis and synthesis module fully supports Block Design Files (**.bdf**) for schematic design entry.

The Quartus II software's Block Editor allows you to create and edit BDFs and open Graphic Design Files (**.gdf**) imported from the MAX+PLUS II software. The Symbol Editor allows you 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 on creating and editing schematic designs, see *Overview: Working with Block & Symbol Editor Files* in Quartus II Help.



Schematic entry methods do not support the synthesis directives or attributes described in this chapter.

Design Flow

The Quartus II design flow using Quartus II integrated synthesis has the following steps:

1. Create a project in the Quartus II software, specifying general project information.
2. Create design files in the Quartus II software, or with a text editor.
3. Specify compiler settings that control the compilation and optimization of the design during synthesis and fitting.
4. Compile the design in the Quartus II software. Basic compilation includes the Analysis & Synthesis, Fitter, Assembler, and Timing Analyzer modules.
5. After obtaining place-and-route results that meet your needs, program the Altera device.

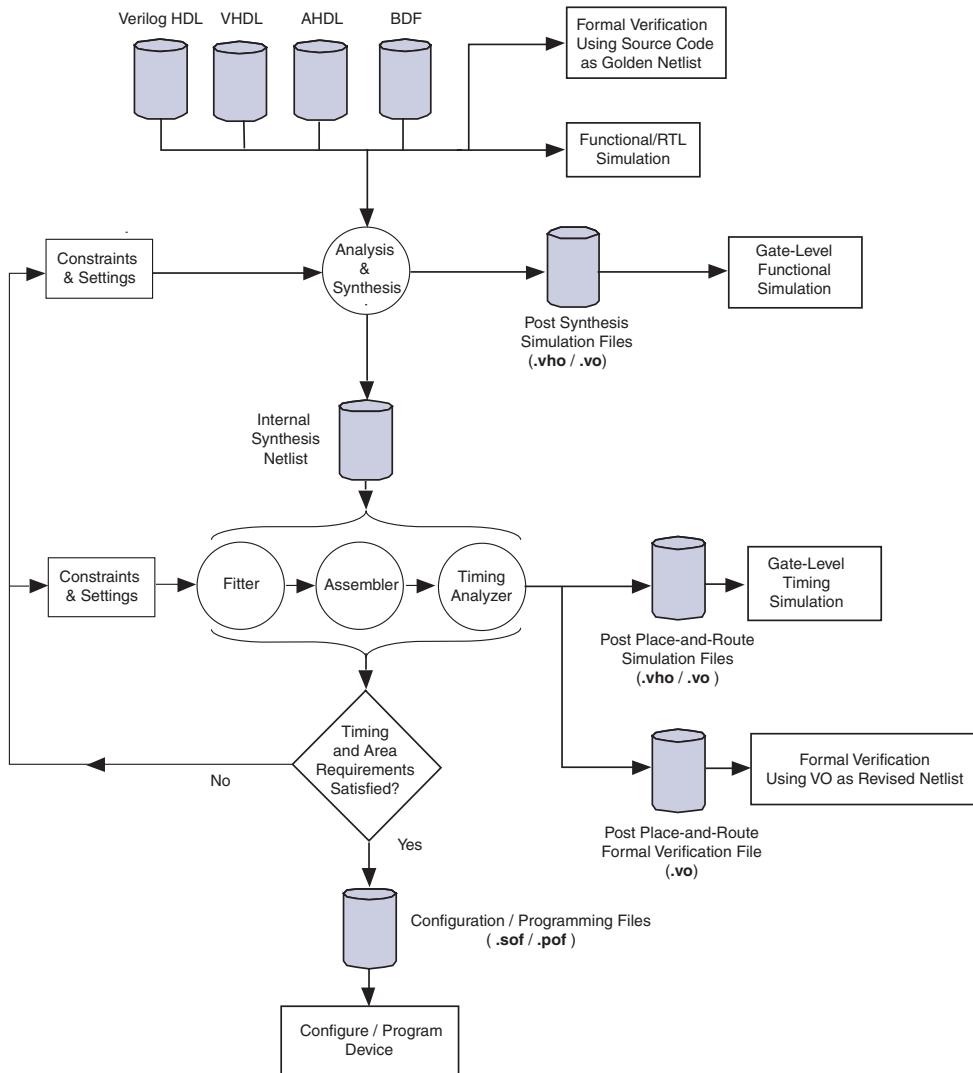
The software provides netlists to allow you to perform functional simulation and gate-level timing simulation in the Quartus II simulator or a third-party simulator, perform timing analysis in a third-party timing analysis tool in addition to the Quartus II Timing Analyzer, and/or perform formal verification in a third-party formal verification tool. The Quartus II software itself offers many other analysis and debug features.



For more information on design flow, and other features in the Quartus II software, refer to Quartus II Help and other chapters in the *Quartus II Handbook*.

Figure 6–1 shows the basic design flow for Quartus II integrated synthesis.

Figure 6–1. Quartus II Design Flow Using Quartus II Integrated Synthesis



Incremental Synthesis

The incremental synthesis feature in the Quartus II software manages a design hierarchy for incremental design by allowing you to divide the design into multiple partitions. Incremental synthesis ensures that when a design is compiled, only those partitions of the design that have been updated will be resynthesized, reducing synthesis time and run-time memory usage. You can change and resynthesize a design partition without affecting other design partitions, which means that node names are maintained during synthesis for all registered and combinational nodes in unchanged partitions.

Conventionally, a hierarchical design is flattened into a single netlist of logic gates before logic synthesis and technology mapping. However, incremental synthesis allows you to partition a hierarchical design along any of its hierarchical boundaries. The individual hierarchical partitions are synthesized and mapped separately by the Quartus II software. In the Quartus II software version 4.2, the hierarchical partitions are then combined—or merged—to form a flattened netlist for further stages of the Quartus II compilation flow, including fitting. The mapped netlist for each partition is stored by the Quartus II software. Therefore, if the source code for one partition changes during the design cycle, only the partition that changed is resynthesized during the next compilation of the design.

Partitions for Incremental Synthesis

A partition represents a portion of the design that you want to synthesize incrementally. Partitions must be bounded by hierarchical boundaries, and therefore, cannot be a portion of the logic within a hierarchical block. When a partition is declared, every hierarchical block within that partition becomes part of the same partition. You can create new partitions for hierarchical blocks within an existing partition, in which case the blocks within the new partition are no longer part of the higher-level partition.

In [Figure 6–2](#), hierarchical blocks B and F form partitions in the complete design, which is made up of blocks A, B, C, D, E, and F. The top-level partition automatically contains the top-level block in the design (i.e., block A in this example) and any logic that is not defined as part of another partition. The design file for the top-level may be just a wrapper for the hierarchical blocks below it, or it may contain its own logic. The shading in [Figure 6–2](#) indicates design partitions. 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 top-level partition A. Another separate partition B contains the logic in blocks B, D, and E.

Figure 6–2. Partitions in a Hierarchical Design

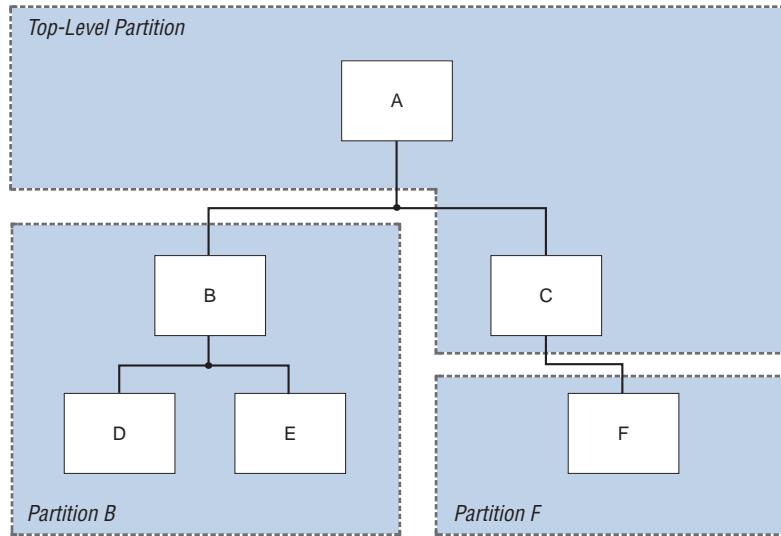
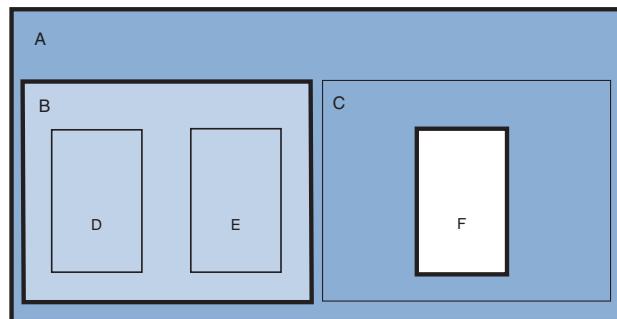


Figure 6–3 shows another representation of the design. The shading indicates the relationship between design blocks and hierarchical partitions.

Figure 6–3. Partitions in a Hierarchical Design Illustrated with Shading



Preparing a Design for Incremental Synthesis

To set up your design for incremental synthesis, turn on incremental synthesis and identify the incremental design partitions, using the following steps:

1. Elaborate the design with **Start > Start Analysis & Elaboration** (Processing menu), or any compilation flow that includes this step. This allows the Quartus II software to identify your design's hierarchy.
2. Identify the partitions in your design by applying the **INCREMENTAL_DESIGN_PARTITION** assignment to the appropriate instances. You can do this using the list of instances under **Compilation Hierarchy** in the **Project Navigator**. Right-click on an instance in the **Project Navigator** and select **Set as Incremental Design Partition** (right button pop-up menu).



An incremental design icon appears next to each instance that is set as a partition.

3. Turn on **Enable incremental synthesis** on the **Compilation Process Settings** page of the **Settings** dialog box (Assignments menu).



When you specify your first partition, a dialog box appears asking whether you wish to enable incremental synthesis. Clicking **Yes** in this dialog box turns on the **Enable incremental synthesis** option.

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

Synthesizing a Design Using Incremental Synthesis

This section describes the two ways to re-synthesize a design with incremental synthesis, using the following features:

- Automatic compilation flow
- Separate synthesis and merge commands

Synthesizing Using the Automatic Compilation Flow

Once incremental synthesis is enabled, it becomes the default compilation procedure under normal circumstances, i.e., when using **Start Compilation** (Processing menu), or by clicking the **Start Compilation** button in the toolbar.

As with the automatic compilation flow, the software synthesizes each partition separately, and then merges the partitions to create a flattened netlist for further stages of the Quartus II compilation flow, including fitting.

For subsequent iterations of analysis and synthesis, the Quartus II software uses the time-stamp on the design files to determine whether the file needs to be resynthesized. The software also checks the values of the following synthesis and netlist optimization settings:

- Optimization technique
- State machine processing
- Perform WYSIWYG primitive resynthesis
- Perform gate-level register retiming

Only settings made globally on the top-level design, or on one of the specified partitions, are checked. Settings made on a level of hierarchy that is not a separate partition, such as block C in [Figure 6–2](#), are not checked.

The software resynthesizes only those partitions that contain changed source code, or changes to one of the synthesis or netlist optimization settings listed above. If the top-level file is modified but no lower-level partitions are affected, only the top-level partition is resynthesized.

The software always maintains the synthesis results for unchanged partitions, and merges newly synthesized partitions with unchanged partitions.

Synthesizing Using the Synthesis & Merge Commands

If you compile your design using the individual compilation steps available from the **Start** menu (Processing menu) and the **Compilation Tool** (Tools menu), instead of using the **Start Compilation** (Processing menu) command, use the separate synthesis and merge commands.

Once incremental synthesis is enabled, you can use **Start >Start Analysis & Synthesis** to separately synthesize each partition. You must then merge the partitions to create a flattened netlist for further stages of the Quartus II compilation flow, including fitting, using the **Start >Start Partition Merge** command (Processing menu).

As with the automatic compilation flow, for subsequent iterations of analysis and synthesis, the Quartus II software uses the time-stamp on the design files to determine whether any file must be resynthesized. The software also checks the following synthesis and netlist optimization settings:

- Optimization Technique
- State Machine Processing
- Perform WYSIWYG primitive resynthesis
- Perform gate-level register retiming

Only settings made globally on the top-level design, or on one of the specified partitions, are checked. Settings made on a level of hierarchy that is not a separate partition, such as block C in [Figure 6–2](#), are not checked.

The software resynthesizes only those partitions that contain changed source code, or changes to one of the synthesis or netlist optimization settings listed above. If the top-level file is modified, but none of the lower-level partitions are affected, only the top-level partition is re-synthesized.

The software always maintains the synthesis results for the unchanged partitions. After each iteration of analysis and synthesis, merge the newly synthesized partitions with the unchanged partitions using the **Start >Start Partition Merge** command (Processing menu).

Forcing Complete Re-synthesis

Because the incremental synthesis flow resynthesizes any partitions that change, most users will not need to force a complete resynthesis of all source files. You may need to do this if you want to resynthesize after making some settings or assignments that do not lead to an automatic resynthesis of a particular partition. Refer to [“Synthesizing Using the Synthesis & Merge Commands” on page 6–11](#) for a list of assignments that do cause resynthesis when the assignments are changed.

If you want to force complete re-synthesis, turn off **Enable incremental synthesis**, re-synthesize your entire design, and then turn on **Enable incremental synthesis** again (if desired). Alternately, to save the extra synthesis run, you can make a small change and save at least one file in each design partition, so that the Quartus II software detects the changes and re-synthesizes each partition on the next analysis and synthesis.

Considerations & Restrictions When Using Incremental Synthesis

To use incremental synthesis effectively, there are some issues to consider when planning your design's structure. Additionally, there are restrictions when using incremental synthesis with other Quartus II features. This section discusses the following considerations:

- Hierarchical considerations
- Restrictions on megafunction partitions
- Resource balancing
- Preserving hierarchical boundary logic option
- Formal verification tools
- Back-annotating node locations or importing/exporting using the Altera LogicLock™ design methodology
- Virtual I/O pins
- SignalTap® II logic analyzer

Hierarchical Considerations

When planning a design, keep in mind the size and scope of each partition, and how likely it is that different parts of your design will change as your design develops.



For guidelines on design hierarchical partitioning, refer to *Hierarchical Design Partitioning* section of the *Design Recommendations for Altera Devices* chapter in Volume 1 of the *Quartus II Handbook*.

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

- Register all inputs and outputs of each block. This makes logic synchronous and avoids any delay penalty on signals that cross between partitions.
- Do not use tri-state signals or bidirectional ports on hierarchical boundaries. 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. Because this requires optimizing through hierarchies, lower-level boundary tri-state signals are not supported with a block-level or incremental design methodology.

Using incremental synthesis, 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 bidirectional ports that feed a bidirectional pin at the top level, all the logic that forms the bidirectional I/O cell must reside in the same partition.

- Remember that logic is not synthesized, or optimized, across partition boundaries, which means any constant value (e.g., signals set to GND) will not be propagated across partitions.

Restrictions on Megafunction Partitions

The Quartus II software does not support partitions for a megafunction instantiation. 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 (i.e., where the software uses 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.

Resource Balancing

You may have to do some manual resource balancing across partitions. When using incremental synthesis, each partition is synthesized separately, with no data about the resources used in other partitions. This means 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.

For example, in the regular 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 (i.e., 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 once 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.

Refer to “[Megafunction Inference Control](#)” on page 6–35 for more information on resource balancing. 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 tips on resource balancing and reducing resource utilization, refer to the appropriate *Resource Utilization Optimization Techniques* section of the *Design Optimization for Altera Devices* chapter in Volume 2 of the *Quartus II Handbook*.

Preserve Hierarchical Boundary Logic Option

The **Preserve Hierarchical Boundary** logic option is available during synthesis of each partition; however, hierarchical boundaries are currently not preserved throughout the rest of the Quartus II compilation design flow after the partitions are merged at the end of incremental synthesis.

If you require that hierarchical boundaries be set to **Firm** or **Relaxed** throughout the Fitter stage, and other compilation stages, use the standard synthesis flow, and do not use incremental synthesis to synthesize your design.

Formal Verification Tools

Formal verification tools typically require that hierarchical blocks be maintained throughout the compilation flow. However, as explained earlier, incremental synthesis design partitions are not currently maintained throughout fitting and other compilation stages.

If your formal verification flow requires that hierarchical boundaries be preserved throughout the Fitter stage, and other compilation stages, use the standard synthesis flow, and do not use incremental synthesis to synthesize your design.

Back-Annotating Node Locations or Importing/Exporting Using the LogicLock Design Methodology

Incremental synthesis preserves node names throughout the synthesis process. However, the Fitter can make changes to node names using optimizations such as register packing and physical synthesis optimizations. Back-annotating placement and importing/exporting location assignments using the LogicLock design methodology requires that node names stay consistent throughout the compilation flow.

If you require fixed node names for logic location back-annotation, use the **Save a node-level netlist into a Verilog Quartus Mapping File** option on the **Compilation Process Settings** page of the **Settings** dialog box (Assignments menu) to generate a Verilog Quartus Mapping (.vqm) file with the post-compilation node names. Writing out a VQM file and using it as the design's source causes the Quartus II software to enforce persistent naming of nodes. Using fixed node names in a Quartus II-generated VQM is not compatible with incremental synthesis; therefore, you must use the standard synthesis flow.



For more information on back-annotating and using the LogicLock design methodology, refer to the *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*.

Virtual I/O Pins

The incremental synthesis design flow does not currently support virtual pins. If you need to use virtual pins with the LogicLock feature for a team-based or other hierarchical methodology, use the standard synthesis flow and do not use incremental synthesis to synthesize your design.

SignalTap II Logic Analyzer

The incremental synthesis design flow does not currently support the ability to insert the SignalTap II logic analyzer into synthesized design partitions.

If you are using the SignalTap II logic analyzer for on-chip debugging, use the standard synthesis flow and do not use incremental synthesis to synthesize your design.

Types of Synthesis Options

The Quartus II software provides a number of options to guide the synthesis process and achieve optimal results. You can use synthesis directives, synthesis attributes, and Quartus II logic options to control synthesis.



Versions of Quartus II software earlier than 2.1 did not support synthesis directives or attributes; the software treated these options as comments. The behavior of the Quartus II software is different if designs compiled in earlier versions of the software included these synthesis options. You may need to change older code to take into account that the software recognizes these options.

This section defines three types of synthesis options: synthesis directives, synthesis attributes, and Quartus II logic options. The following section, “[Quartus II Synthesis Options](#)” on page 6–20, describes the most common and useful of the synthesis options in the Quartus II software, and provides HDL examples of how to use each option where applicable.

Synthesis Directives

The Quartus II software supports synthesis directives, also commonly called pragmas. You can include synthesis directives in Verilog HDL or VHDL code as comments. These directives are not Verilog HDL or VHDL commands; however, synthesis tools use them to control the synthesis process in a particular manner. Other tools such as simulators ignore these directives and treat them as comments.

You can enter synthesis directives in your code using the following syntax, where *<directive>* and *<value>* are variables, and the entry in brackets is optional. The examples in this chapter demonstrate each syntax form.

Synthesis Directives in Verilog HDL

```
// synthesis <directive> [ =<value> ]
or
/* synthesis <directive> [ =<value> ] */
```

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.



Because formal verification tools do not recognize keywords `exemplar` and `pragma`, they are not supported when using formal verification.

Synthesis Attributes

The Quartus II software supports synthesis attributes for Verilog HDL and VHDL, also commonly called pragmas. Synthesis attributes are similar to synthesis directives in that they drive the synthesis process. However, attributes always apply to a specific design element. Some synthesis attributes are also available as Quartus II logic options.

Assignments or settings made with synthesis attributes take precedence over assignments or settings made through the Quartus II user interface, the QSF, or the Tcl interface.

The Verilog-2001 and VHDL language definitions provide specific syntax for specifying attributes. However in Verilog-1995 HDL, you must embed attribute assignments in comments similar to synthesis directives. You can enter attributes in your code using the following syntax, 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.

Synthesis Attributes in Verilog-1995 HDL

```
// synthesis <attribute> [ = <value> ]  
or  
/* synthesis <attribute> [ = <value> ] */
```



Verilog-1995 comment-embedded attributes must be used as a suffix to (i.e., 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, separate the attributes with spaces, as follows:

```
//synthesis <attribute1> [ = <value> ] <attribute2> [ = <value> ]
```

For example, to set the `maxfan` attribute to 16 (See the “[Maximum Fan-Out](#)” section for details) and set the `preserve` attribute (See the “[Preserve Registers](#)” on page [6–25](#) 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.



Because formal verification tools do not recognize keywords `exemplar` and `pragma`, they are not supported when using formal verification.

Synthesis Attributes in Verilog-2001 HDL

```
( * <attribute> [ = <value> ] *)
```



Verilog-2001 attributes must be used as a prefix to (i.e., placed before) a declaration, module item, statement, or port connection, and used as a suffix to (i.e., 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, separate the attributes with commas, as follows:

```
( * <attribute1> [ = <value1> ], <attribute2> [ = <value2> ] *)
```

For example, to set the `maxfan` attribute to 16 (See “[Maximum Fan-Out](#)” section for details) and set the `preserve` attribute (See the “[Preserve Registers](#)” section for details) on a register called `my_reg`, use the following syntax:

```
( * preserve, maxfan = 16 *) reg my_reg;
```

Synthesis Attributes in VHDL

```
attribute <attribute> : <attribute type> ;
attribute <attribute> of <object> : <object type> is <value> ;
```

Quartus II Logic Options

Quartus II logic options control many aspects of the synthesis and place-and-route process. You can set logic options in the Quartus II graphical user interface through the **Assignment Editor** (Assignments menu). Quartus II logic options allow you to set the associated attributes without editing the source HDL code. Logic options can be used with all design entry languages supported by the Quartus II software: Verilog HDL, VHDL, AHDL, and schematic entry.

Quartus II Synthesis Options

This section discusses many common Quartus II synthesis options. These options help you control the synthesis process within the Quartus II software, and can help you achieve the optimal results for your design. Some options are simply synthesis directives, some are only available as either synthesis attributes or logic options, and some are available as both synthesis attributes and logic options.

For information on using other Quartus II synthesis attributes to make pin-related assignments and set other options (that are only available as logic options) in your Verilog HDL or VHDL code, see “[Setting Other Quartus II Options in Your HDL Source Code](#)” on page 6–40.



Because Verilog HDL is case-sensitive, synthesis directives and attributes are also case sensitive.

Translate Off & 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. 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. The following are examples of these directives.

Verilog HDL Example of Translate Off & On

```
// synthesis translate_off
parameter tpd = 2;      // Delay for simulation

#tpd;
// synthesis translate_on
```

VHDL Example of Translate Off & On

```
-- synthesis translate_off
use std.textio.all;
-- synthesis translate_on
```

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 using formal verification.

In the following examples, 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 below.

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
```

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
```

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 logic “don’t care” values. VHDL case statements must be full, so the attribute does not apply.

 Using this attribute on a case statement that is not full avoids the latch inference problems discussed in the *Design Recommendations for Altera Devices* 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, e.g. 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 6–17).

When using the `full_case` attribute, there is a potential cause for 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 the following example 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 the following example 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`.

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];
            2'b11: y=a[3];
endmodule
```

```

2'b01: y=a[1];
2'b10: y=a[2];
endcase
endmodule

```

Verilog-2001 syntax also accepts the following statements in the case header instead of the comment form shown in the example above.

```
(* 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 honor 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 use of the parallel_case attribute, due to the possibility of introducing mismatches between Verilog HDL functional and post-Quartus II simulation.

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, then more than one output (a, b, c) is high as well, a situation that cannot occur in functional HDL simulation.

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 following statements in the `case` (or `casez`) header instead of the comment form shown in the example above.

```
(* parallel_case *) casez (sel)
```

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>~reg0`).

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 below. In this example, 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.

Verilog HDL Example of a Keep Attribute

```
wire my_wire /* synthesis keep = 1 */;
```

Verilog-2001 Example of a Keep Attribute

```
(* keep = 1 *) wire my_wire;
```

VHDL Example of a Keep Attribute

```
signal my_wire: bit;
```

```
attribute syn_keep: boolean;
attribute syn_keep of my_wire: signal is true;
```

Preserve Registers

This attribute and logic option direct 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 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.



The option cannot preserve registers that have no fan-out.

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 below. In this example, 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.

Verilog HDL Example of a Preserve Attribute

```
reg my_reg /* synthesis preserve = 1 */;
```

Verilog-2001 Example of a Preserve Attribute

```
(* preserve = 1 *) reg my_reg;
```

VHDL Example of a Preserve Attribute

```
signal my_reg : stdlogic;  
  
attribute preserve : boolean;  
attribute preserve of my_reg : signal is true;
```



Setting the **Preserve Registers** logic option does not affect registers that are removed during the analysis and elaboration stage of compilation (before logic synthesis). To fully preserve the register throughout compilation, use the HDL attribute instead of the logic option.

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. You can also use this option to reduce the load of critical signals, which can improve performance. You can use this 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. 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, in the third condition above where asynchronous control signals are involved, to avoid the possible situation that small differences in timing could produce functional differences in the implementation. 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.



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 on netlist optimizations, see the *Netlist Optimization & 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, or you can set the `maxfan` attribute in your HDL code as shown below. In this example, 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.

Verilog HDL Example of a MaxFan Attribute

```
reg clk_gen /* synthesis maxfan = 50 */;
```

Verilog-2001 Example of a MaxFan Attribute

```
(* maxfan = 50 *) reg clk_gen;
```

VHDL Example of a MaxFan Attribute

```
signal clk_gen : stdlogic;  
  
attribute maxfan : signal ;  
attribute maxfan of clk_gen : signal is 50;
```

Optimization Technique

This logic option specifies the goal for logic optimization during compilation, i.e., whether to attempt to achieve maximum speed performance or minimum area usage, or a balance between the two.

Table 6–1 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 (Assignments menu).

Table 6–1. 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 6–1:

- (1) 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.

State Machine Processing

This logic option specifies the processing style used to compile a state machine. **Table 6–2** 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 (Assignments menu).

Table 6–2. 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

Table 6–2. State Machine Processing Settings

Setting	Description
One-Hot	Encodes the state machine in the one-hot style
User-Encoded	Encodes the state machine in the manner specified by the user

The default state machine encoding, 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.



See the *Recommended HDL Coding Styles* chapter in the *Quartus II Handbook* for guidelines to ensure that your state machine is inferred and encoded correctly.

In addition, in VHDL designs, the state assignments created automatically by the Quartus II software can be overridden by using specific state assignments with the enum_encoding attribute. The enum_encoding attribute must follow the associated type declaration and precede any associated signal declarations. To use the enum_encoding attribute during compilation, set the **State Machine Processing** logic option to **User-Encoded** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu), or use the **Assignment Editor** (Assignments menu).



For more information, see the *Manually Specifying State Assignments* topic in the Quartus II Help.

Preserve Hierarchical Boundary

This logic option determines how strictly the hierarchical boundaries between design entities should be maintained during logic synthesis. **Table 6–3** lists the settings for the option, which you can only apply to a design entity. Lower-level entities do not inherit their parent entity's setting for this option.

Table 6–3. Preserve Hierarchical Boundary Settings

Setting	Description
Off	Completely ignores boundaries and therefore allows unlimited optimization. This setting provides the greatest logic minimization.
Relaxed	Allows only partial cross-boundary optimization, which may reduce the compilation time. Non-trivial inputs and outputs of the entity are visible during simulation and timing analysis.
Firm	Strictly maintains hierarchical boundaries. This setting may increase compilation time, increase logic cell count, and negatively affect design performance.

The **Relaxed** setting means that the Compiler preserves hierarchical boundaries. However, certain signals such as VCC and GND are propagated and optimized through the boundaries. The **Firm** setting does not allow optimization across boundaries, and keeps each hierarchical block separate.



When a formal verification tool is selected in on the **EDA Tool Settings** page of the **Settings** dialog box (Assignment menu), this option is automatically set to firm for entities that contain block-boxes for formal verification.

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 logic elements (LEs) or adaptive logic modules (ALMs). This option is available for Stratix® II, Stratix, Stratix GX, Cyclone™ II, Cyclone, and MAX II devices.

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 "if", "case", or "? : ". 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_VECTORS in VHDL. The **Restructure Multiplexers** option identifies buses of multiplexer trees that have a similar structure. When 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 6–4 lists the settings for the logic option, which you can only apply to a design entity. You can also set this option for your whole project on the **Analysis & Synthesis Settings** page in the **Settings** dialog box (Assignments menu).

Table 6–4. 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 , and Off when the Optimization Technique option is Balanced or Speed . (Note that since the default Optimization Technique is Balanced for many device families including Stratix and Stratix II devices, this option is turned on by default for those families.)

Once 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 6–5 describes the information that is listed in the **Multiplexer Restructuring Statistics** report table for each bus of multiplexers.

Table 6–5. Multiplexer Information in the Multiplexer Restructuring Statistics Report	
Heading	Description
Multiplexer Inputs	The number of different choices being 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 on optimizing for multiplexers, refer to the *Multiplexers* section of the *Design Recommendations for Altera Devices* 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). You can apply this option to any register or to a pin with the logic configurations described below:

- 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

For the register to power up to with the specified logic level, the Compiler may perform NOT gate push-back on the register.

Power-Up Don't Care

This logic option causes registers to power up with a “don't care” logic level (X), or the logic level most appropriate for the design. This option allows the Compiler to change the power-up condition of a register to, for example, minimize your design's area usage. This option is turned on by default.

For example, a register may have its D input tied to VCC. If you turn this option off, the register powers up low even though it goes high at the first clock signal. If you turn this option on, the Compiler sets the power-up value of the register to high and, therefore, can eliminate the register and connect the output of the register to VCC. If the Compiler performs this type of optimization, 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**.



Versions of the Quartus II software earlier than version 2.1 did not include this option. If you compile an older design that relies on registers to power-up to a specific level, the Compiler may synthesize the design differently. Turn off the **Power-Up Don't Care** option if you want your design to use the power-up behavior of older versions of Quartus II software.

Remove Duplicate Logic

If you turn on this option, the Compiler removes logic that is identical to other logic in the design. If two functions generate the same logic, the Compiler removes the second one, and the first one fans out to the second one's destinations. Additionally, if the deleted logic function has different logic option assignments, the Compiler ignores them. This option is turned on by default.

When turned on, this option also removes all duplicate registers in the same way as the **Remove Duplicate Registers** option. If you do not want the Compiler to remove certain registers when this option is turned on, turn off the **Remove Duplicate Registers** option for those registers. See [Table 6–6](#) for more details.

Even if you turn this option on, the Compiler does not remove duplicate logic that you inserted deliberately. If a function's output feeds an LCELL buffer, the Compiler always treats it as a unique signal and the **Remove Duplicate Logic** option does not apply (i.e., the Compiler does not remove an LCELL buffer if you turn on this option).

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.

The Compiler recognizes this option only if you turned on the **Remove Duplicate Logic** option. When turned on, the **Remove Duplicate Logic** option also removes duplicate registers. Therefore, you should use this option only if you want to prevent the Compiler from removing duplicate registers that you have used deliberately. That is, you should use this option only with the **Off** setting. See [Table 6–6](#). You can apply this option to an individual register or a design entity that contains registers.

Table 6–6. Settings for Remove Duplicate Logic & Remove Duplicate Registers		
Remove Duplicate Logic Setting	Remove Duplicate Registers Setting	Description
On (Default)	On (Default)	Removes logic (including registers) if it is identical to other logic in the design.
On	Off	Preserves all registers for which the Remove Duplicate Registers option is turned off. Removes logic (including any other registers) if it is identical to other logic in the design.
Off	On or Off	Preserves duplicate logic and registers.

Remove Redundant Logic Cells

This logic option removes redundant LCELL primitives or WYSIWYG cells. If you turn on this option, the Compiler optimizes a circuit for area and speed. The project-wide option is turned off by default.

Megafunction Inference Control

The Quartus II Compiler automatically recognizes certain types of HDL code and infers the appropriate megafunction when a megafunction provides optimal results. That is, the software uses the Altera megafunction code when compiling your design even though you did not specifically instantiate the megafunction. The software infers megafunctions resulting in logic that is optimized for Altera devices. The area and/or 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 elements.



For details on coding style recommendations when targeting megafunctions in Altera devices, see 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 & 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 (Assignment menu), or disable the option for a specific block with the **Assignment Editor** (Assignments menu).



Any registers that the software maps to the `altsync*_accum` and `altsync*_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 (Assignments menu), 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 (Assignment menu), or disable the option for a specific block with the **Assignment Editor** (Assignments menu).

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 black-box 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 explicitly instantiating RAM blocks in separate modules or entities so that as much logic as possible can be verified by the formal verification tool.

RAM Style

This attribute specifies the type of TriMatrix™ embedded memory block that the Compiler should use when implementing an inferred RAM, and is only supported for device families with TriMatrix embedded memory blocks.

The `ramstyle` attribute takes a single string value (in quotation marks) to specify the type of memory block: "M512", "M4K", or "M-RAM". In Verilog HDL, set the `ramstyle` attribute on the declaration of the multidimensional variable that represents an inferred RAM. In VHDL, set the `ramstyle` attribute on a signal or variable declaration that represents an inferred RAM.



In addition to `ramstyle`, the Quartus II software supports the `syn_ramstyle` attribute name for compatibility with other synthesis tools.

The following examples specify that the inferred ram `my_ram` should be implemented using an M512 memory block.

Verilog-1995 Example of a ramstyle Attribute

```
reg [0:7] my_ram[0:63] /* synthesis ramstyle = "M512" */;
```

Verilog-2001 Example of a ramstyle Attribute

```
(* ramstyle = "M512" *) reg [0:7] my_ram[0:63];
```

VHDL Example of a ramstyle Attribute

```
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 "M512";
```

Multiplier Style

The `multstyle` attribute specifies the implementation style for multiplication operations (*) in your HDL source code. Using this attribute, you can specify whether you prefer the Compiler to implement a multiplication operation in general logic or dedicated hardware, if available in the target device.



Specifying a `multstyle` of "dsp" does not guarantee that the Quartus II software can implement a multiplication in dedicated hardware. The final implementation depends, among other things, on the availability of dedicated hardware in the target device, the size of the operands, and whether or not one or both operands are constant.

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.



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, the `multstyle` attribute directs the Quartus II software to implement all multiplications inside module `my_module` in dedicated multiplication hardware.

Verilog-1995 Example of Applying a multstyle Attribute to a Module Declaration

```
module my_module (...) /* synthesis multstyle = "dsp" */;
```

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.

For example, in the following code, the `multstyle` attribute applied to variable `result` directs the Quartus II software to implement $a * b$ in general logic rather than dedicated hardware.

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
```

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 target variable or enclosing module. For example, in the following code, the `multstyle` attribute indicates that $a * b$ should be implemented in dedicated hardware.

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 can not 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. For example, in the following code, the `multstyle` attribute directs the Quartus II software to use dedicated hardware, if possible, for all multiplications inside architecture `rtl` of entity `my_entity`.

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. For example, in the following code, the `multstyle` attribute associated with signal `result` directs the Quartus II software to implement `a * b` in general logic rather than dedicated hardware.

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;
```

Setting Other Quartus II Options in Your HDL Source Code

This section describes Quartus II synthesis attributes that can be used to set other Quartus II options and settings in your HDL source code. The attributes described in the “[Chip Pin](#)” and “[Use I/O Flip-Flops or Registers](#)” sections can help you make pin-related assignments in your HDL code, and the attribute described in the “[Altera Attribute](#)” section can be used to make any other Quartus II option or setting assignments in your HDL code. Assignments made with these synthesis attributes take precedence over assignments made through the Quartus II user interface, the [QSF](#), and the [Tcl](#) interface.

Use I/O Flip-Flops or Registers

This attribute directs the Quartus II software to implement input, output, and output enable flip-flops (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](#) (Assignments menu).



For more information on which device families support fast input, output, and output enable registers, refer to your device family data sheet or handbook or to 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 the following examples, 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.

Verilog HDL Example of a 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 following type of statements instead of the comment form shown in the example above.

```
(* useioff = 1 *)    input [1:0] a, b;
(* useioff = 1 *)    output [2:0] o;
```

VHDL Example of a useioff Attribute

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity top_level 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 top_level;

architecture rtl of top_level is
    signal o_reg, a_reg, b_reg : unsigned(1 downto 0);

begin
    process(clk)
    begin
        a_reg <= a;
        b_reg <= b;
        o_reg <= a_reg + b_reg;
    end process;

    o <= o_reg;
end rtl;
```

Altera Attribute

This attribute enables you to apply Quartus II options and assignments to an object (entity, instance, or net) in your HDL source code. With `altera_attribute`, you can control synthesis options from your HDL source even when the options lack a specific HDL synthesis attribute (like many of the logic options presented earlier in this chapter). You can also use this attribute to pass option settings and assignments to phases of the Compiler flow beyond Analysis and Synthesis, such as Fitting.

Assignments and settings made with the Altera Attribute take precedence over assignments and settings made through the Quartus II user interface, the QSF, or the Tcl interface.

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 6-17](#).

The attribute value is a single string containing a list of QSF variable assignments separated by semicolons, as follows:

```
"-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 (matching the syntax of the QSF) for each QSF variable assignment.

```
-from <source> -to <target> -section_id <section>
-name <variable> <value>
```

The syntax for the full attribute value, including the optional target, source, and section tags for two different QSF assignments is shown in the following example:

```
" [-from <source_1>] [-to <target_1>] [-section_id
<section_1>] -name <variable_1> <value_1>; [-from <source_2>
[-to <target_2>] [-section_id <section_2>] -name <variable_2>
<value_2>]"
```

If a variable's assigned value is a string of text, you must use escaped quotes around the value, 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 QSF 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.

The following examples 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 automatically moves the attribute to the inferred instance.

Verilog-1995 Example of Applying Altera Attribute to an Instance

```
reg my_reg /* synthesis altera_attribute = "-name
POWER_UP_LEVEL HIGH" */;
```

Verilog-2001 Example of Applying Altera Attribute to an Instance

```
(* altera_attribute = "-name POWER_UP_LEVEL HIGH" *)
reg my_reg;
```

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";
```

The following examples 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.

Verilog-1995 Example of Applying Altera Attribute to an Entity

```
module my_entity(...) /* synthesis altera_attribute =
"-name AUTO_SHIFT_REGISTER_RECOGNITION OFF" */;
```

Verilog-2001 Example of Applying Altera Attribute to an Entity

```
(* altera_attribute = "-name
AUTO_SHIFT_REGISTER_RECOGNITION OFF" *) module
my_entity(...);
```

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;
```

Chip Pin

This attribute enables you to assign pins to the ports of an entity or module in your HDL source. You may only assign pins 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.

The following examples show different ways of assigning input pin my_pin1 to Pin C1 and my_pin2 to Pin 4 on a target device.

Verilog-1995 Example of Applying Chip Pin to a Single Pin

```
input my_pin1 /* synthesis chip_pin = "C1" */;
input my_pin2 /* synthesis altera_chip_pin_lc = "@4" */;
```

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;
```

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.

The following examples assign my_pin[2] to Pin_4, my_pin[1] to Pin_5, and my_pin[0] to Pin_6.

Verilog-1995 Example of Applying Chip Pin to a Bus of Pins

```
input [2:0] my_pin /* synthesis chip_pin = "4, 5, 6" */;
```

The following example 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.

```
input [0:2] my_pin /* synthesis chip_pin = "4, ,6" */;
```

The following example assigns my_pin[2] to Pin 4 and my_pin[0] to Pin 6, but leaves my_pin[1] unassigned.

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";
```

Node-Naming Conventions in Quartus II Integrated Synthesis

Finding the logic node names after synthesis can be helpful during verification or while debugging a design. This section provides an overview of the conventions used by the Quartus II software when naming the codes 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, the 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. The names for other nodes, particularly LEs or ALMs that contain only combinational logic, can change due to logic optimizations that the software performs.



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.

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 with 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.

You can turn off the **Display Entity Name for Node Name** option on the **Compilation Process Settings** page of the **Settings** dialog box (Assignment menu) to instruct the Compiler to generate node names that do not contain the name for each level of the hierarchy. With this option on, the node names use the following convention:

| <instance_name 0> | <instance_name 1> | . . . | <instance_name n>

Node-Naming Conventions for Registers (DFF or D Flip-Flop Atoms)

In Verilog HDL and VHDL, inferred registers are named after the `reg` or `signal` connected to the output.

For example, the following is a description of a register in Verilog HDL that creates a DFF primitive called `my_dff_out`:

```
wire dff_in, my_dff_out, clk;  
  
always @ (posedge clk)  
    my_dff_out <= dff_in;
```

Similarly, the following is a description of a register in VHDL that creates a DFF primitive called `my_dff_out`:

```
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 above examples) is also an output pin of your top-level design, the Quartus II software cannot use that name for the register (e.g., 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 following Verilog HDL code produces a register called `q~reg0`:

```
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, then the port is removed during hierarchy flattening and the register retains its original name, in this case `q`.

Registers That Can Change 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 its name 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 issues that affect register names:

- State machines
- Inferred counters, adder-subtractors, shift registers, memory, and DSP functions
- Input and output registers of RAM and DSP blocks

State Machines

If a state machine is inferred from your HDL code, then the registers that represent the states will be 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 AHD L, 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 Counters, Adder-Subtractors, Shift Registers, Memory, & DSP Functions

The Quartus II software infers megafunctions from Verilog HDL and VHDL code for logic that forms counters, adder-subtractors, shift registers, RAM, ROM, and arithmetic functions that can be placed in DSP blocks.



For information on inferring megafunctions, refer to the *Recommended HDL Coding Styles* chapter in the *Quartus II Handbook*.

Because counters and adder-subtractors are part of a megafunction instead of generic logic, the combinational logic and registers exist 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.

Input & Output Registers of RAM & 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 on packing registers into RAM and DSP megafunctions, refer to the *Recommended HDL Coding Styles* chapter in the *Quartus II Handbook*.

Node-Naming Conventions for Combinational Logic Cells

Whenever possible for Verilog HDL, VHDL, and AHD L 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 following Verilog HDL code, where Quartus II integrated synthesis uses the names “c”, “d”, “e” and “f” for the combinational logic cells that are produced.

```
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.

If logic cells, such as those created in the above example, 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 the Stratix II and Cyclone II 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 above. 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.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script.

For detailed information about specific scripting command options and Tcl API packages, type `quartus_sh --qhelp` at a system command prompt to run the Quartus II Command-Line and Tcl API Help utility.



For more information on Quartus II scripting support, including examples, refer to the *Tcl Scripting* and *Command-Line Scripting* chapters 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>\n    -to <Instance Name>
```

Quartus II Synthesis Options

Table 6–7. Quartus II Synthesis Options (Part 1 of 2)

Setting Name	QSF Variable	Values	Type
Implement as Output of Logic Cell	IMPLEMENT_AS_OUTPUT_OF_LOGIC_CELL	ON, OFF	Instance
Preserve Registers	PRESERVE_REGISTER	ON, OFF	Instance
Maximum Fanout	MAX_FANOUT	<Maximum Fan-out Value>	Instance
State Machine Processing	STATE_MACHINE_PROCESSING	AUTO, "MINIMAL BITS", "ONE HOT", "USER-ENCODED"	Global, Instance
Optimization Technique	<device family>_OPTIMIZATION_TECHNIQUE	Area, Speed, Balanced	Global, Instance

Setting Name	QSF Variable	Values	Type
Implement as Output of Logic Cell	IMPLEMENT_AS_OUTPUT_OF_LOGIC_CELL	ON, OFF	Instance
Preserve Registers	PRESERVE_REGISTER	ON, OFF	Instance
Maximum Fanout	MAX_FANOUT	<Maximum Fan-out Value>	Instance
State Machine Processing	STATE_MACHINE_PROCESSING	AUTO, "MINIMAL BITS", "ONE HOT", "USER-ENCODED"	Global, Instance
Optimization Technique	<device family>_OPTIMIZATION_TECHNIQUE	Area, Speed, Balanced	Global, Instance

Table 6–7. Quartus II Synthesis Options (Part 2 of 2)			
Setting Name	QSF Variable	Values	Type
Power-Up Level	POWER_UP_LEVEL	HIGH, LOW	Instance
Preserve Hierarchical Boundary	PRESERVE_HIERARCHICAL_BOUNDARY	Off, Relaxed, Firm	Instance
Restructure Multiplexers	MUX_RESTRUCTURE	On, Off, Auto	Global, Instance
Power-Up Don't Care	ALLOW_POWER_UP_DONT_CARE	ON, OFF	Global
Remove Duplicate Registers	REMOVE_DUPLICATE_REGISTERS	ON, OFF	Global, Instance
Remove Duplicate Logic	REMOVE_DUPLICATE_LOGIC	ON, OFF	Global, Instance
Remove Redundant Logic Cells	REMOVE_REDUNDANT_LOGIC_CELLS	ON, OFF	Global
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
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
Auto Shift-Register Replacement	AUTO_SHIFT_REGISTER_RECOGNITION	ON, OFF	Global, Instance
Allow Any Shift Register Size for Recognition	ALLOW_ANY_SHIFT_REGISTER_SIZE_FOR_RECOGNITION	ON, OFF	Global, Instance
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

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 up to IOBANK_n, where n is the number of I/O banks in a particular device.

Preparing a Design for Incremental Synthesis

To set up your design for incremental synthesis, enable incremental synthesis and identify the incremental design partitions.

Identify the incremental design partitions on an instance level using the following Tcl command:

```
set_instance_assignment \
-name INCREMENTAL_DESIGN_PARTITION <Partition Name> \
-to <Hierarchical Instance Name>
```

For information on the hierarchical instance name, refer to “[Hierarchical Node-Naming Conventions](#)” on page [6-46](#).

It is also possible to use entity assignments for INCREMENTAL_DESIGN_PARTITION, as in the following command:

```
set_global_assignment \
-name INCREMENTAL_DESIGN_PARTITION <Partition Name> \
-entity <Design Entity Name>
```

Using an entity assignment means that all instances of that entity will become a partition and will use the same netlist in the complete design. This is recommended only if you have a complete understanding of the instantiations of the design entity. Do not use this type of assignment if different instances of the entity are customized with parameters or different settings. Doing so can result in incorrect synthesis results because the software selects one of the implementations for both instances of the entity.

Turn on incremental synthesis using the following Tcl command:

```
set_global_assignment -name \
ENABLE_INCREMENTAL_SYNTHESIS ON
```

Synthesizing a Design Using Incremental Synthesis

This section discusses the two ways to use incremental synthesis.

Synthesizing Using the Automatic Compilation Flow

Once incremental synthesis is enabled in the QSF file, or through a Tcl command, incremental synthesis automatically occurs when you compile using the `execute_flow -compile` command for the `quartus_sh` compiler executable.

Synthesizing Using the Synthesis & Merge Commands

Use the separate synthesis and merge commands if you compile your design using the individual compiler executables (e.g., `quartus_map` and `quartus_fit`) instead of using the `execute_flow -compile` command for the `quartus_sh` compiler executable.

For example, to enable incremental synthesis when using the `quartus_map` executable, perform the following two steps:

1. Type the following command at a command prompt:

```
quartus_map --incr_synth ↵
```

2. Merge the synthesized partitions to create a flattened netlist for further stages of the Quartus II compilation flow, including fitting. Type the following command:

```
quartus_cdb -merge ↵
```

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.

qii51009-2.1

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 key design flows, methodologies, and techniques for achieving good performance in Altera® devices using the Synplicity Synplify and Synplify Pro software with the Quartus® II software, including the following:

- General design flow with the Synplify and Quartus II software
- Synplify optimization strategies, including timing-driven compilation settings, optimization options, and Altera-specific attributes
- Exporting designs to the Quartus II software using NativeLink® integration
- Cross-probing with the Quartus II software
- Guidelines for Altera megafunctions and LPM Functions, instantiating them in a clear box or black box flow using the MegaWizard® Plug-In manager and tips for inferring them from hardware description language (HDL) code
- Block-based design with the Quartus II LogicLock™ methodology, including the Synplify Pro Multipoint flow

This chapter assumes that you have set up, licensed, and are familiar with the Synplify or Synplify Pro software.

The content in this chapter applies to both the Synplify and Synplify Pro software unless otherwise specified.



To obtain and license the Synplify software, and for more information on using the software, see the Synplicity web site at www.synplicity.com.

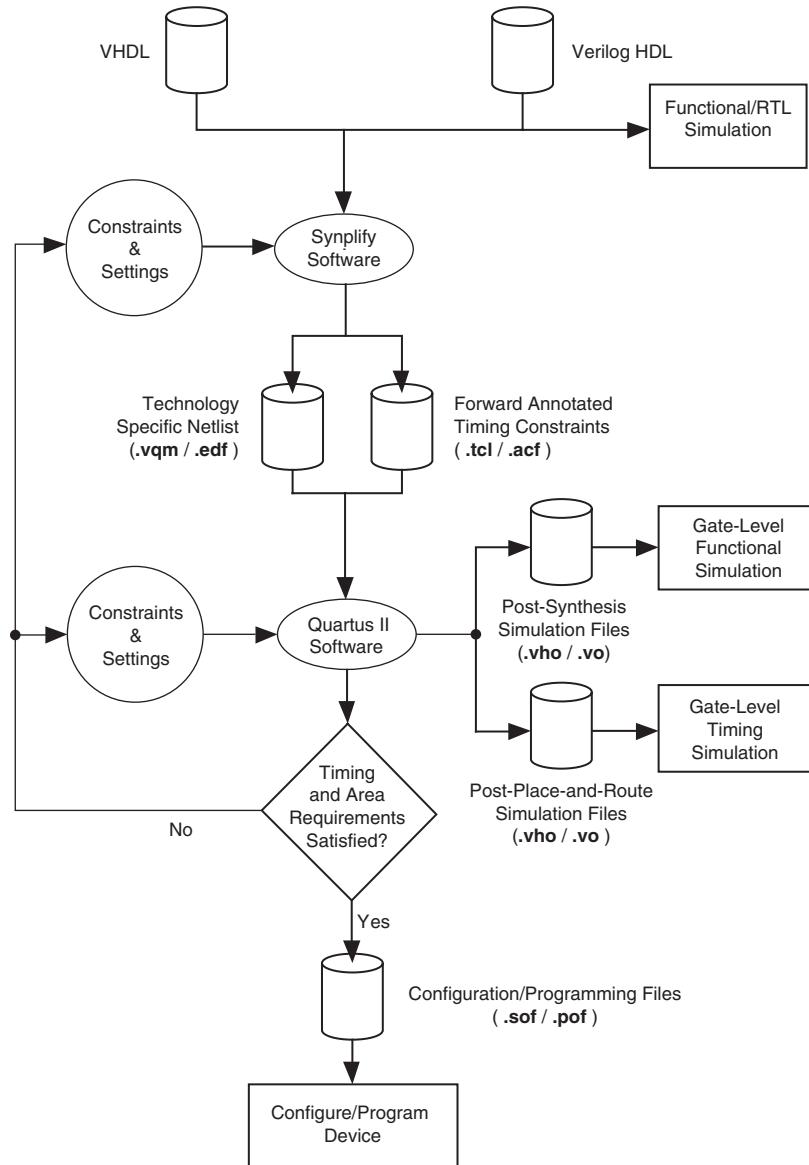
Design Flow

The basic steps in a Quartus II design flow using the Synplify software are the following:

1. Create Verilog HDL or VHDL design files in the Quartus II design software, in the Synplify software, or with a text editor.
2. Set up a project and add the HDL design files in the Synplify software 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 netlist and the tool command language (Tcl) constraint file generated by the Synplify software to 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 Quartus II software.

Figure 7-1. Recommended Design Flow

The Synplify and Synplify Pro software tools support both VHDL and Verilog HDL source files. Synplify Pro 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 editor 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 in the Tcl file for use by the Quartus II software. You can save all project options and included files in a Synplify project file (**.pj**).

The HDL Analyst 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. Use the HDL Analyst to visually analyze and debug the design. The HDL Analyst supports cross probing between the RTL and Technology views, the HDL source code, and the Finite State Machine (FSM) viewer. See “[Finite State Machine \(FSM\) Compiler](#)” on page [7-9](#).

 A separate license file is required to enable the HDL Analyst in the Synplify software. The Synplify Pro software comes with the HDL Analyst.

Once synthesis is complete, import the EDIF or 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.

If the area and timing requirements are satisfied, use the files generated from 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 Quartus II software and re-run the synthesis. Repeat the process until the area and timing requirements are met.

While simulation may be performed at various points in the process, detailed 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 on 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 on netlist optimizations, see the *Netlist Optimizations and Physical Synthesis* chapter in Volume 2 of the *Quartus II Handbook*.

In some cases, source code may also need modification 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 files with a short description of each file.

Table 7–1. Synplify Intermediate & Output Files	
File Extensions	File Description
.srs	Technology-independent RTL netlist that can be read only by Synplify
.srm	Technology view netlist
.srr (1)	Synthesis report file
.edf/.vqm (2)	Technology-specific netlist in electronic design interchange format (EDIF) (.edf) or Verilog Quartus Mapping (.vqm) file format
.acf/.tcl (3)	Forward-annotated constraints file containing constraints and assignments

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, as it is the only reliable source of timing information. This report file includes post-synthesis device resource utilization statistics which 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 only for ACEX® 1K, FLEX® 10K, FLEX 10KA, FLEX 10KE, FLEX 6000, FLEX 8000, MAX® 7000, MAX 9000, and MAX 3000 devices. A Verilog Quartus Mapping (.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+PLUS® II software. A Tcl file (.tcl) for the Quartus II software is created for all devices, which also contains Tcl commands to create a Quartus II project and, if applicable, the MAX+PLUS II assignments are imported from the ACF file.

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 obtain the required results.



For additional design and optimization techniques, see the *Design Recommendations for Altera Devices* chapter in Volume 1 and the *Design Optimization for Altera Designs* 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 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 on applying the attributes discussed in this section, see the “Adding Attributes and Directives” section in the *Tasks and Tips* chapter of the *Synplify User Guide*.

Implementations in Synplify Pro

Use the **New Implementation** command (Project menu) in the Synplify Pro software to create different synthesis results without overwriting the others. 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 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 through 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, such as clock frequencies, multi-cycle paths, and false paths, that are applied in the Synplify software to be forward-annotated for the Quartus II software using a Tcl script file for timing-driven place-and-route.



The Synplify synthesis report file (.srr) contains timing reports of estimated place-and-route delays. Altera's Quartus II software can perform further optimizations on a post-synthesis netlist from a synthesis vendor such as Synplicity. In addition, designs may contain black boxes or IP functions that have not been optimized by the third-party synthesis software. Actual timing results are only obtained after the design has gone through full place-and-route 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 a constraint file (.sdc) with the SCOPE editor in the Synplify software.

Use the SCOPE editor to set global frequency requirements for the entire design and individual clock settings. Use the **Clocks** tab in the SCOPE editor 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 and 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. By default, all clocks are in different groups, so paths with different registers using more than one clock signal are not analyzed. 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 editor 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 editor 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. The following equations illustrate the relationship between T_{CO}/T_{SU} and the input/output delays:

$$T_{CO} = \text{clock period} - \text{external output delay}$$

$$T_{SU} = \text{clock period} - \text{external input delay}$$

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.

Multi-Cycle Paths

Specify any multi-cycle paths in the design in the **Multi-Cycle Paths** tab of the SCOPE editor or with the `define_multicycle_path` attribute. A multi-cycle path is one that requires more than one clock cycle to propagate. It is important to specify which paths are multi-cycle to avoid having the Quartus II and 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 editor or with the `define_false_path` attribute.

Finite State Machine (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` compiler 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 this 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 provide the best performance and shortest clock-to-output delays. However, one-hot implementations are usually larger than binary implementations.
Safe	Generate 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.

The example below, “[VHDL Code for `syn_encoding`](#)”, shows sample VHDL code for applying the `syn_encoding` directive.

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 automatically explore different encoding styles for a state machine 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.

General Optimization Attributes & Options

The following sections list other options that you can modify in the Synplify software to affect your design performance.

Maximum Fan-out

When dealing with critical path nets with high fan-outs, 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 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 (see the “[Register Packing](#)” on page [7-11](#) section).

Preserving Nets

During synthesis, the compiler maintains ports, registers, and instantiated components. However, some nets may not be maintained in order 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.

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 can be applied to modules/architectures. Setting the value to “hard” maintains the boundaries of a module/architecture and prevent 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.

Retiming in Synplify Pro

The Synplify Pro software can retime a design. Retiming can improve the timing performance of sequential circuits by moving registers (register balancing) across combinational elements. Be aware that retimed registers incur name changes. Turn on the retiming option in the **Device** tab in the **Implementation Options** section or by using the `syn_allow_retimming` attribute.

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 can be applied to inputs and outputs. This attribute is not supported for any of the MAX device families. [Figure 7–2](#) shows how to set the attribute in the SCOPE editor.

Figure 7–2. altera_chip_pin_lc with SCOPE Editor

	Enabled	Object Type	Object	Attribute	Value	Val Type	Description
1	<input checked="" type="checkbox"/>		data_out	altera_chip_pin_lc	14,5,16,15	string	I/O pin location
2	<input checked="" type="checkbox"/>						
3	<input checked="" type="checkbox"/>						

The “[altera_chip_pin_lc with VHDL for ACEX 1K and FLEX 10KE Devices](#)” example 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.

altera_chip_pin_lc with VHDL for ACEX 1K and FLEX 10KE Devices

```
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 “[altera_chip_pin_lc with Other Devices](#)” example 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.

altera_chip_pin_lc with Other Devices

```
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

Use either of these attributes to implement logic in ESB/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 inputs/outputs 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® architecture is not supported by this option. For designs targeting devices that do not have ESB/EABs, this attribute is ignored).

altera_io_powerup

Use this attribute to define the power-up value of an I/O register which has no set or reset. The attribute takes a string value ("high | low") and can be applied to ports that have I/O registers.

altera_io_opendrain

Use this attribute to specify open-drain mode I/O ports. The attribute takes a Boolean value and can be applied to outputs or bidirectional ports for devices that support open-drain mode.

Exporting Designs to the Quartus II Software Using NativeLink Integration

After a design is synthesized in the Synplify software, a VQM (or EDIF) file 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 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 graphical user interface (GUI) interface 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.

Running the Quartus II Software from within the Synplify Software

To use the Quartus II software from within the Synplify software, follow the steps below:

1. 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.
2. Choose one of the following commands from the **Quartus II** submenu under the **Options** menu in the Synplify software:
 - a. **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 then configure options for the project and execute any Quartus II commands.
 - b. **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. The `<project_name>.tcl` file contains device, timing, and location assignments.

Using the Quartus II Software to Launch the Synplify Software

You can set up the Quartus II software to run the Synplify software for synthesis using NativeLink integration.



For detailed information on using NativeLink integration with the Synplify software, go to *Specifying EDA Tool Settings* in the Quartus II Help index.



Running the Synplify software with NativeLink integration requires a floating network license (as opposed to a node-locked single-PC license), because batch mode compilation is supported only with floating licenses.

You can also import the results of the Synplify synthesis and use them from within the Quartus II software. Among other methods, a Quartus II project can be created and compiled by running the `<project_name>_cons.tcl` script file. This is done by running the following Tcl command in the Tcl Console:

```
source <project_name>_cons.tcl ↵
```



To open the Tcl Console, select **Utility Windows > Tcl Console** (View menu) in the Quartus II software.

Cross-Probing with the Quartus II Software

The Quartus II and Synplify software support bidirectional cross-probing in the Windows operating system environment. With cross-probing, selecting an object in one application highlights the same object in the other. This feature thus provides the ability to connect post-place-and-route timing results to the source code. Cross-probing is supported for all Altera devices that generate a VQM netlist when compiled in the Synplify software (an EDIF netlist is generated instead of a VQM for designs targeting ACEX 1K, FLEX 10K, FLEX 6000, MAX 7000, and MAX 3000 devices). The cross-probing capability provides a truly integrated flow between your front-end and back-end EDA tools and reduces debugging time.

Some examples of cross-probing uses include the following:

- NativeLink integration allows you to cross-probe to the Synplicity HDL Analyst viewer when selecting a node in the Quartus II Floorplan. From the HDL Analyst, you can cross-probe to the source code that generated the post-synthesis nodes.
- Selecting an AND primitive in the HDL Analyst RTL view highlights the corresponding logic elements in the Quartus II Floorplan so that you can find the location where it is being placed in the Altera device.
- A critical path in the Quartus II message window and in the Quartus II Timing Analyzer can be cross-probed to the source code in the Synplicity synthesis tools with the Quartus II Floorplan.
- You can cross-probe from the Synplicity synthesis tools to the Quartus II Floorplan and view the placement and timing for state machines or view the routing of high fan-out nodes.

Enabling Cross-Probing

You must enable cross-probing in both applications. In order to activate the cross-probing capability in Synplicity's synthesis tools and the Quartus II software, both tools must be open and have the design or project loaded.

To enable cross-probing in the Synplify software, open a schematic view, and select **External Cross Probing Engaged** (HDL Analyst menu).

To enable cross-probing in the Quartus II software, turn on **Enable cross-probing between Quartus II and other EDA tools** option on the **EDA Tool Options** page under **General** in the **Options** dialog box (Tools menu).

The Synplify and the Quartus II software interface with each other through a process called **xprobe_server.exe**. From the Quartus II Floorplan and the Synplify HDL Analyst, the nodes can be further probed internally within the respective tools.

Cross-Probing from the Quartus II Software

To perform cross-probing from the Quartus II software, highlight the desired nodes in the Quartus II Floorplan. When you highlight the objects in the Quartus II Floorplan, they are simultaneously highlighted in the Technology view of the Synplify HDL Analyst.

To cross-probe from the Quartus II message window, right-click on the appropriate message in the messages window and select **Locate**. This highlights the appropriate nodes in the Quartus II Floorplan and in the Synplify HDL Analyst.

To locate critical paths of timing violations by cross-probing from the Quartus II Timing Analyzer, right-click an entry in the Quartus II Timing Analyzer and select **Locate in Timing Closure Floorplan**. This highlights the appropriate nodes in the Quartus II Floorplan and in the Synplify HDL Analyst.

Cross-Probing from the Synplify Software

To perform cross-probing from Synplify software, open the HDL Analyst view and select **Technology and Flattened View** (HDL Analyst menu) in the Synplify software. Highlight the objects you want to cross-probe in the Quartus II software. When objects are highlighted in the HDL Analyst, they are simultaneously highlighted in the Quartus II Floorplan and any open HDL Analyst window.

You can locate to the source code in the Synplify software from the HDL Analyst by double-clicking the selected node. If the VHDL or Verilog HDL source file is not open, the Synplify software automatically opens the file.

You can also cross-probe from the Synplify software source code to the HDL Analyst RTL view by selecting **RTL** and **Flattened View (HDL Analyst)** menu in the Synplify software. Highlight the desired source code in the Synplify software by right-clicking and selecting **Highlight in Analyst**.

In the Synplify Pro software, you can cross-probe from the Synplify Pro timing report or log file. To do this, open the HDL Analyst RTL view and highlight the appropriate text in the Synplify Pro text editor. Right-click and choose **Select Port/Net/Instance**.

Guidelines for Altera Megafunctions & Architecture-Specific Features

Altera provides parameterizable megafunctions including the 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 inferring them from generic HDL code.

If you want to instantiate a megafunction in your HDL code, you can do so by using 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 7-18 describes the MegaWizard Plug-In Manager flow with the Synplify software.



For more information on specific Altera megafunctions, see the Quartus II Help. For more information on IP functions, consult 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 7-22.



For a detailed discussion on instantiating vs. inferring megafunctions, see 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 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 either creates a VHDL or Verilog HDL wrapper file that instantiates the megafunction (a black box methodology), or for some megafunctions can generate a fully synthesizable netlist for improved results with EDA synthesis tools such as Synplify (a clear box methodology). Both clear box and black box methodologies are described in the following sections.

Clear Box Methodology

Using the MegaWizard Plug-In Manager-generated fully synthesizable netlist is referred to as a clear box methodology because the Synplify software can “see” into the megafunction file. The clear box feature enables the synthesis tool to report more accurate timing estimates and resource utilization, and take better advantage of timing driven optimization than a black box methodology.

This clear box feature is enabled by turning on the **Generate clear box body (for EDA tools only)** option in the **MegaWizard Plug-In Manager** (Tools menu) for certain megafunctions. If the option does not appear, then clear box models are not supported for the selected megafunction. The Synplify software supports clear box models for Stratix and Cyclone™ devices. Turning this option on causes the Quartus II MegaWizard Plug-In Manager to generate a synthesizable clear box netlist instead of the megafunction wrapper file described in the “[Black Box Methodology](#)” on page [7–19](#).

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 Synplify project. Also include the `stratix.v` library file from the `lib/altera` directory of the Synplify installation directory; this file provides the port and parameter definitions of the clear box primitives. Finally, include the megafunction clear box netlist file, `<output file>.v`, along with your Synplify-generated VQM netlist in your Quartus II project.

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 Synplify project. Finally, include the megafunction clear box netlist file, `<output file>.vhd`, along with your Synplify-generated VQM netlist in your Quartus II project.

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 Synplify software. The black box wrapper file is generated by default in the **MegaWizard Plug-In Manager** (Tools menu) and is available for all megafunctions.

The black-box methodology does not allow the synthesis tool any visibility into the function module 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. See “[Other Synplify Software Attributes for Black-Boxing](#)” on page [7-21](#) for details.

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.

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 Black-Boxing](#)” on page 7-21.

“[Top-Level Verilog HDL Code with Black Box Instantiation of lpm_counter](#)” below 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.

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 syn attribute below is added to
// black box 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.

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 section [Other Synplify Software Attributes for Black-boxing](#).

“[Top-level VHDL Code with Black Box Instantiation of lpm_counter](#)” below 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.

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 Black-Boxing

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 with a “gray box” methodology by adding the `syn_tpd`, `syn_tsu`, and `syn_tco` attributes. See the following “[Verilog HDL Example](#)” section for a Verilog HDL example.

Verilog HDL Example

```
module ram32x4(z,d,addr,we,clk);
/* synthesis syn_black_box syn_tco1="clk->z[3:0]=4.0"
   syn_tpdl="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
```

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 on applying these attributes, see the “Adding Attributes and Directives” section in the *Tasks and Tips* chapter of the *Synplify User Guide*.

Inferring Altera Megafunctions from HDL Code

The Synplify software uses Behavior Extraction Synthesis Technology (B.E.S.T.) algorithms to infer high-level structures such as RAMs, ROMs, operators, FSMs, etc. It then keeps the structures abstract for as long as possible in the synthesis process. This allows for 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.

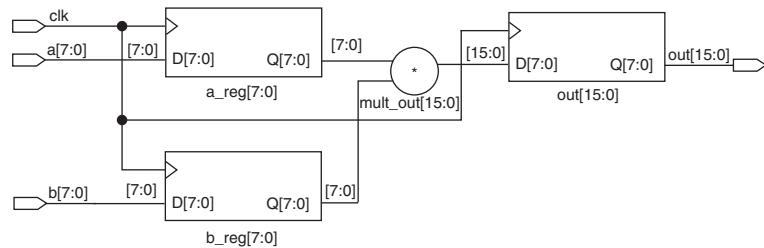


For coding style recommendations and examples for inferring megafunctions in Altera devices, see the *Recommended HDL Coding Styles* chapter in Volume 1 of the *Quartus II Handbook*.

Multiplication

Figure 7–3 shows the RTL 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 **1pm_mult** megafunction. For devices with DSP blocks, the software may implement the **1pm_mult** function in a DSP block instead of LEs, depending on device utilization.

Figure 7-3. HDL Analyst View of lpm_mult Megafunction (Unsigned 8x8 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 includes 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 (LEs or 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 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 certain Altera devices. The user 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 7–3 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 7–3. Attribute Settings for DSP Block in the Synplify Software		
Attribute Name	Value	Description
syn_multstyle	lpm_mult	LPM Function inferred and multipliers implemented in DSP block
syn_multstyle	logic	LPM function not inferred and multipliers implemented LEs by the Synplify software

The following examples show simple Verilog HDL and VHDL code using the `syn_multstyle` attribute.

Signal Attributes for Controlling DSP Block Inference in Verilog HDL

```
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
```

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;
```

RAM

Follow the guidelines below for the Synplify software to successfully infer RAM in a design:

- The address line must be at least 2 bits wide.
- Resets on the memory are not supported. Refer to the device family documentation for information on 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`. 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 Stratix II, Stratix, Cyclone II, and Cyclone device designs, 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.

[“VHDL Code for Inferred Dual-Port RAM”](#) below shows sample VHDL code for inferring dual-port RAM.

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;

```

The “[VHDL Code for Inferred Dual-Port RAM Preventing Bypass Logic](#)” example 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.

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;

```

Inferring ROM

Follow the guidelines below for the Synplify software to successfully infer ROM in a design:

- The address line must be at least 2 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

Block-Based Design with the Quartus II LogicLock Methodology

As designs become more complex and designers work in teams, a block-based hierarchical design flow is often an effective design approach. In this approach, you perform optimization on individual sub-blocks and each sub-block has its own output netlist file. After you optimize all of the sub-blocks, you integrate them into a final design and optimize it at the top level.

You can use the Synplify software with the LogicLock design methodology in the Quartus II software to perform block-based or team-based compilation. The Synplify Pro software also offers the MultiPoint Synthesis feature to provide an incremental synthesis flow with the LogicLock design methodology.

MultiPoint synthesis, which is available for certain device technologies in the Synplify Pro software, provides an automated incremental synthesis flow and can reduce runtime. The MultiPoint feature manages a design hierarchy to let you design incrementally and synthesize designs that take too long for top-down synthesis. MultiPoint synthesis allows different netlist files to be created for different sections of a design hierarchy, supporting the LogicLock design methodology. 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. A designer can change and resynthesize their section of a design without affecting other sections of a design.

You can also create different netlist files manually with the Synplify software (basic Synplify and Synplify Pro). 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 resynthesized when you compile the design. You can make changes, optimize and resynthesize your section of a design without affecting other sections.

Using the LogicLock design methodology, you can place each block's logic into a fixed or floating region in an Altera device. You then have the opportunity to maintain the placement and the performance of your

blocks in the Altera device. If all the netlists are contained in one Quartus II project, you can use the LogicLock flow to back-annotate the logic within the other regions. In this case, when you recompile with one new VQM netlist file, the placement and assignments for unchanged netlist files assigned to different LogicLock regions are not affected. Therefore, one designer can make changes to a piece of code that exists in an independent block and not interfere with another designer's changes, even if all the blocks are integrated in a top-level design. With the LogicLock design methodology, separate pieces of a design can progress from development to testing without affecting other areas of a design.



For more information on using the LogicLock feature in the Quartus II software, see *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*.

Hierarchy & 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 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 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 design methodology. You should use tri-state drivers only at the external output pins of the device and at the top-level block in the hierarchy.

Creating a Design with Multiple VQM Files

The first stage of a hierarchical design flow is to generate multiple VQM files, enabling you to take advantage of the LogicLock incremental design flow and the incremental fitter in the Quartus II software. If the whole design is in one VQM file, changes in one block affect other blocks because of possible node name changes.

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 black-boxing each block that you want to be part of a LogicLock region.

In the Multipoint synthesis flow (Synplify Pro only), you create multiple VQMs from one easy-to-manage top-level synthesis project. Using the manual black-boxing 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 need to create the appropriate Quartus II projects to place and route the design.

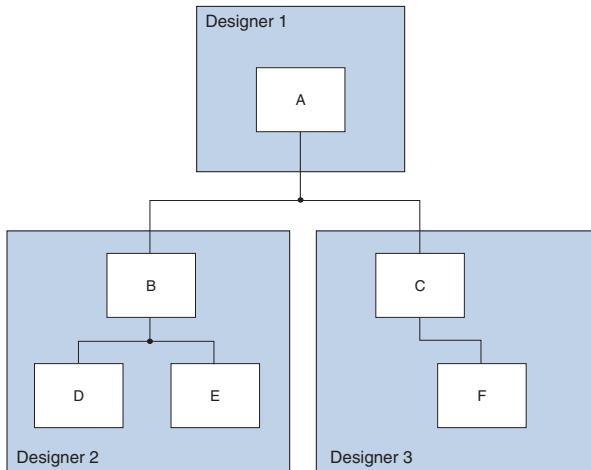
Creating a Design with Multiple VQM Files Using Synplify Pro Multipoint Synthesis

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 create LogicLock regions.

Set Compile Points & 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 block 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.

Figure 7–4 shows an example of a design hierarchy that can be split into multiple compile points.

Figure 7-4. Design Hierarchy

In this case, modules A, B, and C are compile points, and there is a separate netlist file for each block.

Compile points are optimized in isolation from their parent, which could 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 stands on its own, 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 that contain them. When you apply the appropriate 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.

Compile points are applied to the module or architecture in the Synplify Pro SCOPE spreadsheet or the constraint file (.sdc). You cannot set a compile point in the Verilog/VHDL source code. You can set the constraints manually using Tcl or by editing the SDC file. You can also use the graphical user interface (GUI) which provides two methods, manual or automated as shown below.

Defining Compile Points Using Tcl or SDC

To set compile points using Tcl and an SDC file, use 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.

```
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 which 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 that you need to do to set up each compile point. To use the automated method:

1. Select **New** under the File menu and choose to create a new **Constraint File**, or click the **SCOPE** icon in the tool bar. Select **Compile Point** from the **Select File Type** tab of the **Create a New SCOPE File** dialog box.
2. 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 the LogicLock design flow, 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 will be. Constraints are not automatically budgeted, so manual time budgeting is essential.
- 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.
- You must put tri-state buffers in the top-level module because tri-state drivers are located at the outputs of Altera devices. Tri-state buffers coded in lower-level files do not get pushed to the top-level automatically.
- There is no incremental synthesis support for top-level logic; any logic in the top-level is resynthesized during every run.



For further details about compile points, see the *Synplify Pro User Guide and Reference Manual* at www.synplicity.com/literature/index.html.

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 used with the LogicLock design methodology. 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 both 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, 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.

Table 7–4 shows the valid combinations of the LogicLock attributes.

Table 7–4. 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 appropriate region size and location.
Floating	Fixed	Assumes size of LogicLock constraint area is already optimal in 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 module. Attributes can be set in the attribute tab of the SCOPE spreadsheet.

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 regarding these attributes, see the Synplify Pro online help or reference manual.

During compilation, the Synplify Pro software creates a *<top-level project>.tcl* file that provides the Quartus II software with the appropriate 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. This example is for module A (instance `u1`) in the project named `top` where the region name `cpl1_1` was selected by Synplify Pro for the compile point.

```
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 on Tcl commands, see the *Tcl Scripting* chapter in Volume 2 of the *Quartus II Handbook*.

Creating a Quartus II Project for Multiple VQM Files

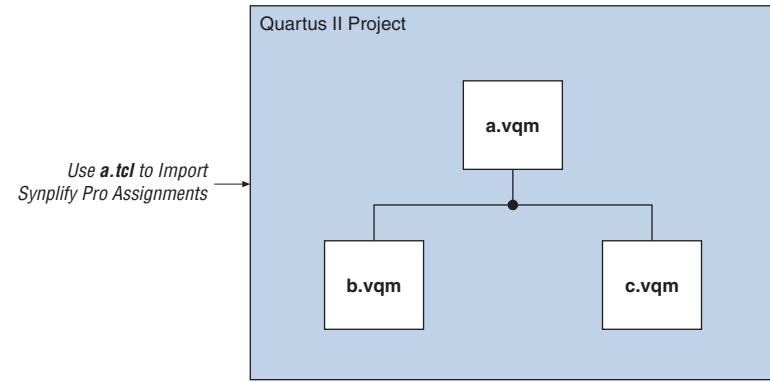
You can use the following methods to import the VQM files into the Quartus II software.

- Use the *<top-level project>.tcl* file that contains the Synplify Pro assignments for all blocks within the project. This method allows the top-level designer to import all the blocks into one Quartus II project for an incremental flow. You can optimize all modules within the project at once. [Figure 7–5](#) shows a visual representation of the design flow.



If additional optimization is required for individual blocks, each designer can take their VQM file and create a separate Quartus II project at that time with the appropriate assignments. New assignments would then have to be added to the top-level project through the LogicLock import function.

Figure 7–5. Design Flow Using Multiple VQM Files with One Quartus II Project



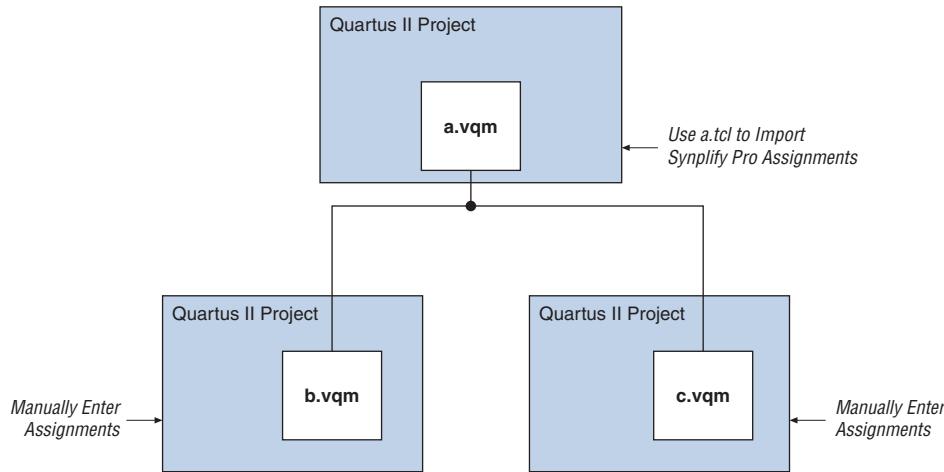
- Generate multiple Quartus II projects, one for each block in the design. Each designer in the project can optimize their block separately within the Quartus II software and back-annotate their

blocks. Figure 7–6 shows a visual representation of the design flow. The optimized sub-designs can be brought into one top-level Quartus II project using the LogicLock import function.



Each designer must enter their assignments into the Quartus II software manually because Synplify Pro doesn't create a Tcl file for the lower-level modules.

Figure 7–6. Design Flow Using Multiple VQM Files with Multiple Quartus II Projects



For more information on importing LogicLock assignments, see the *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*.

Generating a Design with Multiple VQM Files Using Black Boxes

This section describes how to manually generate multiple VQM files using a black boxing technique. The following manual flow was supported in previous versions of the Synplify Pro software, and is discussed here because some designers or teams may want more control over the project for each submodule. In addition, 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 module and top-level design that you want to maintain as a separate VQM 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, follow these general guidelines.

For lower-level modules, perform these steps:

1. Turn on **Disable I/O Insertion** for the target technology in the **Implementation Options** dialog box.
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 editor.
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.

For top-level designs, follow these steps:

1. Turn off **Disable I/O Insertion** for the target technology.
2. Read the HDL files for top-level designs.
3. Black-box lower-level modules in the top-level design.
4. Add constraints with the SCOPE constraint editor.
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 block-boxing modules in a block-based and team-based design flow. [Figure 7–4 on page 7–30](#) shows an example of a design hierarchy that is split up as a team-based design.

In [Figure 7–4 on page 7–30](#), 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. 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 VQM files:

1. Generate a VQM file for module B. Use **B.v/.vhdl**, **D.v/.vhdl**, and **E.v/.vhdl** as the source files.
2. Generate a VQM file for module C. Use **C.v/.vhdl** and **F.v/.vhdl** as the source files.
3. Generate a top-level VQM file **A.v/.vhdl** for module A. Ensure that you black box modules B and C, which were optimized separately in the previous steps.

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, are treated as a black box by the software. Use the `syn_black_box` attribute to indicate that you intended to black-box the given module. In Verilog HDL, you must provide an empty module declaration for the module that is treated as a black box.

The “[Black-Boxing Example for Top-Level File A.v](#)” example below 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.

Black-Boxing Example 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));
    C 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 C follow here.
// These module declarations (including ports) are required for black boxing.

module B (data_in, clk, ld, data_out) /*synthesis syn_black_box */ ;
    input data_in, clk, ld;
    output [15:0] data_out;
endmodule

module C (d, clk, e, q) /*synthesis syn_black_box */ ;
    input [15:0] d;
    input clk, e;
    output q;
endmodule
```

```
    output [15:0] q;
endmodule
```

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, are treated as a black box by the software. Use the `syn_black_box` attribute to indicate that you intended to black-box the given component. 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 similarly forwarded to the VQM. To prevent case-sensitive mismatches between VQM, use the same capitalization for black-box and entity declarations in VHDL designs.

“[Black-Boxing Example for Top-Level File A.vhd](#)” shows an example of the `A.vhd` top-level file. If any lower-level files also contain a black-boxed lower-level file in the next level of hierarchy, follow the same procedure.

Black-Boxing Example 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 C 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 C: component is true;

-- Other component declarations in A.vhd go here
```

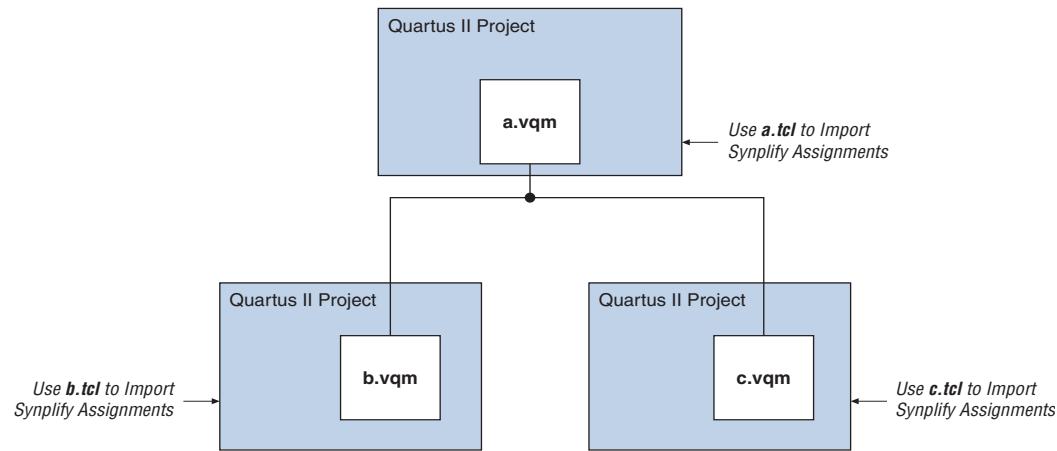
```
signal cnt_out : INTEGER RANGE 0 TO 15;  
  
BEGIN  
  
U1 : B  
PORT MAP (  
    data_in => data_in,  
    clk => clk,  
    ld => ld,  
    d_out => cnt_out );  
  
U2 : C  
PORT MAP (  
    d => cnt_out,  
    clk => clk,  
    e => e,  
    q => data_out );  
  
-- Any other code in A.vhd goes here  
  
END a_arch;
```

After you have completed the steps described in this section, you will have a different VQM netlist file for each block of code. These files can now be used in the LogicLock incremental design methodology 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, providing the Quartus II software with the information to set up a project. Altera recommends the following method for bringing each VQM and corresponding Tcl file into the Quartus II software.

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 back-annotate their blocks. [Figure 7–7](#) shows a visual representation of the design flow. 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 be back-annotated and brought into one top-level project.

Figure 7–7. Design Flow Using Multiple Synplify Projects & Multiple Quartus II Projects

For more information on creating and importing LogicLock assignments, see the *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*.

Conclusion

Advanced synthesis is an important part of the design flow. Taking advantage of the Synplicity Synplify and 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 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.

qii51010-2.1

Introduction

As programmable logic becomes more complex and require increased performance, advanced synthesis has become an important part of the design flow. Combining hardware description language (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.

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, see the Mentor Graphics web site at www.mentor.com. For information on installing the LeonardoSpectrum software and setting up your working environment, see the *LeonardoSpectrum Installation Guide* and the *LeonardoSpectrum User's Manual*.

Design Flow

The basic steps in a LeonardoSpectrum-Quartus II design flow are as follows:

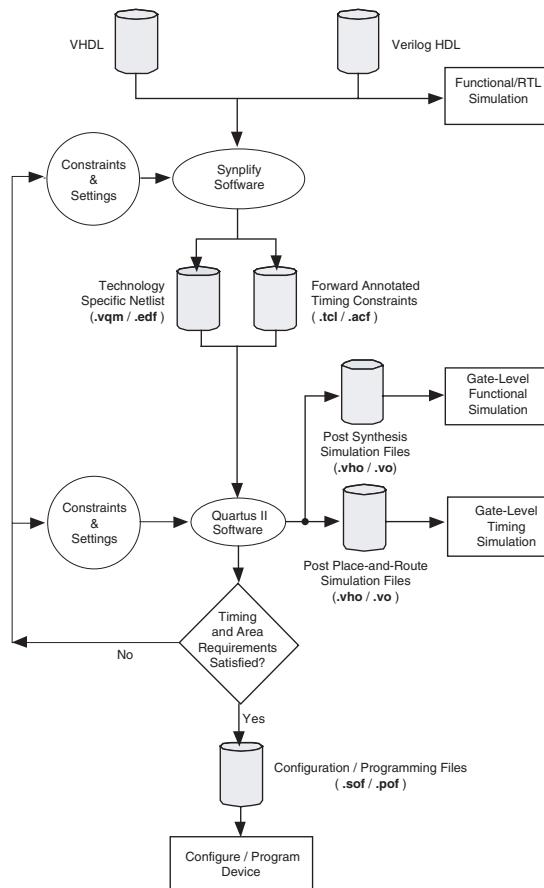
1. Create Verilog HDL or VHDL design files in the LeonardoSpectrum software or a text editor.
2. Import the Verilog HDL or VHDL design files into the LeonardoSpectrum software for synthesis.
3. Select a target device and add timing constraints and compiler directives to help optimize the design during 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 tool command language (.tcl) file 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 8–1 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 8–1**, 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 8–1. Recommended Design Flow Using LeonardoSpectrum & 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 8–1** lists these file extensions with a short description of each file.

Table 8–1. LeonardoSpectrum Intermediate & 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 8–1:

- (1) An assignment and configuration (.acf) file is created only for ACEX® 1K, FLEX® 10K, FLEX 6000, FLEX 8000, MAX® 7000, MAX 9000, and MAX 3000 devices. The ACF is generated for backward compatibility with the MAX+PLUS® II software. A Tcl (.tcl) file 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 does 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 HDL Inventor™ 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 “[Global Power Tab](#)”, “[Clock Power Tab](#)”, and “[Input & Output Power Tabs](#)” 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 can be set 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 & 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 8–2 shows the state machine encoding styles supported by the LeonardoSpectrum software.

Table 8–2. 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 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 8–2](#) shows an example of a LeonardoSpectrum compilation report.

Figure 8–2. 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. See “[Integration with the Quartus II Software](#)” on page 8–9 for more information.

Exporting Designs Using NativeLink Integration

You can use NativeLink® integration to integrate the LeonardoSpectrum software and Quartus II software with a single graphical user interface (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.

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. Choose **Utility > Tcl Console** (View menu), to Open the Quartus II Tcl Console.
3. Type `source <path>/<project name>.tcl ↵`, at a **Tcl Console** command prompt.
4. Choose **Open Project** (File menu), to open the new project, and start compilation by choosing **Start Compilation** (Processing menu).

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, select **Place and Route Path** >**Tools** (Options menu) and specify the location of the Quartus II software executable file (browse to *<Quartus II software installation directory>/bin*).

Guidelines for Altera Megafunctions & 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 (AMPPsm).



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 on 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 8-3 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”. You can use the **Variable Editor** (Tools menu) 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 8-3. Inferring RAM Summary

	Stratix II, Stratix, Stratix GX, Cyclone II, & Cyclone	APEX 20K, APEX 20KE, APEX II, Excalibur, & Mercury	FLEX 10KE & 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, Stratix® memory must be synchronous to be inferred.

To disable ROM inference, set the `extract_rom` variable to “false.” You can use the **Variable Editor** (Tools menu) to enter the value “false” when synthesizing in the user interface with the Advanced Flow tabs, or add the commands `set extract_rom false` to your synthesis script.

Inferring Multipliers & 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 on inferring multipliers and DSP functions, including examples of VHDL and Verilog HDL code, see the *Recommended HDL Coding Styles* chapter in Volume 1 of *The Quartus II Handbook*.

Simple Multipliers

The `1pm_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 `altsmult_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 `altsmult_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 `altsmult_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, MACs, 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 8–4](#), 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 8–4. 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 8–4:

- (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.

```
//synthesis attribute <module name> extract_mac <value>
```

The following Verilog HDL and VHDL codes samples show how to use the extract_mac attribute.

Using Module Level Attributes in Verilog HDL Code

```
module mult_add ( dataaa, datab, dataac, datad, result);
//synthesis attribute mult_add extract_mac FALSE
// Port Declaration
input [15:0] dataaa;
input [15:0] datab;
input [15:0] dataac;
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 = dataaa * `signed datab;
//synthesis attribute mult0_result preserve_signal TRUE

assign mult1_result = dataac * datad;

// This adder can go into the one-level adder in a DSP
// block
assign result = (mult0_result + mult1_result);

endmodule
```

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 8–5 shows the supported values for the dedicated_mult attribute.

Table 8–5. 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 Quartus II maps the multipliers automatically to either the DSP block or logic based on resource availability

Note to Table 8–5:

- (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 `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.

Following are samples of Verilog HDL and VHDL codes, respectively, using the `dedicated_mult` attribute.

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
```

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 $(\text{width_a} + \text{width_b})$. The maximum width of the accumulator can be $(\text{width_a} + \text{width_b} + 16)$. 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 LogicLock Methodology

The LogicLock™ block-based design flow enables users to design, optimize, and lock down a design one section at a time. With the LogicLock methodology, 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 LogicLock design methodology in the Altera Quartus II software, you can partition a new design into a hierarchy of netlist files during synthesis in the Mentor Graphics 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 on hierarchical design methodologies and block-based design flows, see the *Hierarchical Block-Based & Team Based Design Flows* chapter in Volume 1 of the *Quartus II Handbook*.

Hierarchy & Design Considerations

You must plan your design's structure and partitioning carefully to use the 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, see 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 (i.e., 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, see the LeonardoSpectrum on-line documentation by choosing **Open Manuals Bookcase** (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 LogicLock incremental design flow 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, follow these 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, follow these steps:

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 **Clicking Read** in 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 (**Modules** section).
7. Turn on the **LogicLock** option.
8. Type your 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 re-create 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 1.

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 in the LogicLock incremental design flow 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 following commands 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.

```
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 on Tcl commands, see the *TCL Scripting* chapter in Volume 2 of the *Quartus II Handbook*.

You can use the following methods to import the 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 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 back-annotate their blocks. Altera recommends this method for incremental and hierarchical design methodology because it allows each block in the design to be treated separately; each block can be back-annotated and brought into one top-level project with the LogicLock 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. 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 with the LogicLock import function.

In both methods, you can use the steps below 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. Open the Quartus II **Tcl Console** by choosing **Utility Windows Tcl Console** (View menu).
3. Type source *<path>/<project name>.tcl* ↵, see [Figure 8–3](#).

Figure 8–3. Tcl Console Window with Source Command



4. Open the new completed project by choosing **Open Project** (File menu), browse to the project name, and click **Open**.



For more information on 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, follow these 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

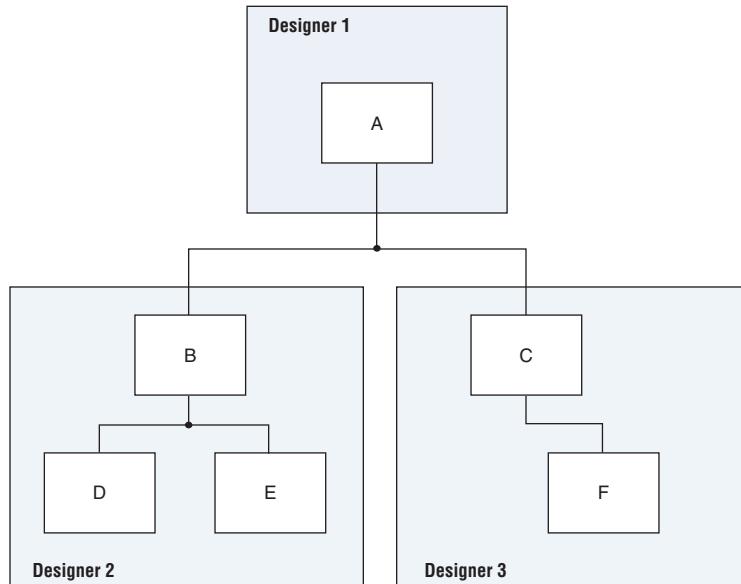
For the top-level design:

- 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 8-4](#), 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 8-4. Block-Based & 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:

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 the previous steps.

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.

The **A.v Top-Level File Black-Boxing Example** section 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.

A.v 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.

The “[A.vhd Top-Level File Black-Boxing Example](#)” 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.

A.vhd 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 in the LogicLock incremental design methodology 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 back-annotate their blocks. 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 incremental and hierarchical

design methodology because it allows each block in the design to be treated separately; each block can be back-annotated and brought 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. 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 LogicLock import function.



For more information on importing LogicLock regions, see the *LogicLock Design Methodology* chapter in the Volume 2 of the *Quartus II Handbook*.

In both methods, you can use the steps below to create the Quartus II project and compile the design:

1. Place the EDIF and Tcl files in the same directory.
2. Open the Quartus II **Tcl Console** by choosing **Utility Windows Tcl Console** (View menu).
3. At a Tcl prompt, type `source <path>/<project name>.tcl ↵`.
4. Open the new project by choosing **Open Project** (File menu), browse to the project name, and click **Open**.
5. Create LogicLock assignments using the **LogicLock Regions** window (Assignments menu).
6. Choose **Start Compilation** (Processing menu).

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. They are modules that contained a LogicLock assignment in the original compilation.



Obtain LeonardoSpectrum's path for each of these modules 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.

The following sample script 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.

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 8–30, 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> ↵
```

You can also run the script from the interface, choose **Run Script** (File menu), 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 and the LogicLock block-based design feature 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.

qii51011-2.1

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 key design methodologies and techniques for achieving good performance in Altera® devices using the Mentor Graphics® Precision RTL Synthesis and Quartus® II software design flow. It includes the following sections:

- General design flow with the Precision RTL Synthesis and Quartus II software
- Creating a project and compiling the design
- Setting constraints to achieve optimal results
- Synthesizing the design and evaluating the results
- Exporting designs to the Quartus II software using NativeLink® integration
- Guidelines for Altera megafunctions and the library of parameterized modules (LPM) functions, instantiating them in a clear box or black box flow using the MegaWizard® Plug-In manager, and tips for inferring them from HDL code
- Block-based design with the Quartus II LogicLock™ methodology

This chapter assumes that you have installed and licensed the Precision RTL Synthesis and Quartus II software.



To obtain and license the Precision RTL Synthesis software, see the Mentor Graphics web site at www.mentor.com. For information on installing the Precision RTL Synthesis software, starting the software, and setting up your working environment, see the *Precision RTL Synthesis User's Manual* in the Mentor Graphics web site.

Design Flow

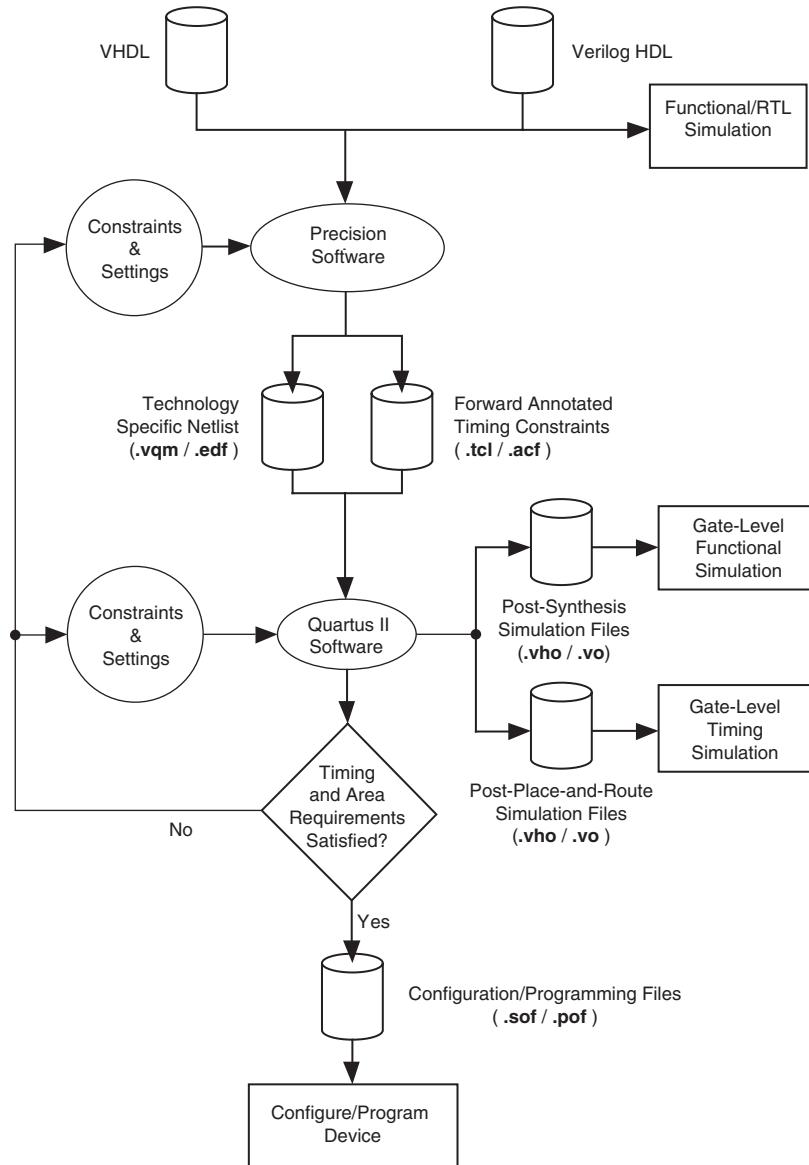
The basic steps in a Quartus II design flow using the Precision RTL Synthesis software are as follows:

1. Create Verilog HDL or VHDL design files in the Quartus II design software or 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, selects your target device, and sets global constraints.

3. Compile the project in the Precision RTL Synthesis software.
4. Add specific timing constraints and compiler directives to optimize the design during synthesis.
5. Synthesize the project in the Precision RTL Synthesis software.
6. Create a Quartus II project and import the technology-specific EDIF (.edf) netlist and the Tcl (.tcl) file generated by the Precision RTL Synthesis software into the Quartus II software for placement and routing, and for performance evaluation.
7. After obtaining place-and-route results that meet your needs, configure or program the Altera device.

These steps are described in more detail in the following sections.

Figure 9–1 shows the design flow described in the steps above.

Figure 9–1. Recommended Design Flow

As shown in [Figure 9–1](#), if your area or timing requirements are not met, you can change the constraints in the Precision RTL Synthesis software or Quartus II software and rerun the synthesis. Repeat the process until the area and timing requirements are met.

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 EDIF netlist in the Quartus II software.



For information on netlist optimizations, see the *Netlist Optimizations and Physical Synthesis* chapter in Volume 2 of the *Quartus II Handbook*. For more recommendations on how to optimize your design, see the *Design Optimization for Altera Devices* chapter in Volume 2 of the *Quartus II Handbook*.

While simulation may be performed at various points in the process, detailed 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 9–1](#) lists those files with a short description of each file type.

Table 9–1. Precision RTL Synthesis Intermediate & Output Files

File Extension(s)	File Description
.sdc	Design Constraints file in Synopsys Design Constraints format
.psp	Precision RTL Synthesis project file
.xdb	Design database file in Mentor Graphics file format
.rep <i>(1)</i>	Synthesis area and timing report files
.edf	Technology-specific netlist in electronic design interchange format (EDIF)
.acf/.tcl <i>(2)</i>	Forward-annotated constraints file containing constraints and assignments

Notes to Table 9–1:

- (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. Use the device utilization reported by the Quartus II software after place-and-route for final resource utilization results. See the “[Synthesizing the Design & Evaluating the Results](#)” section 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 .acf is generated for backward compatibility with the MAX+PLUS® II software. A Tcl file for the Quartus II software is created for all devices, which also contains Tcl commands to create and compile a Quartus II project.

Creating a Project & Compiling the Design

After creating your design files, create a project in the Precision RTL Synthesis software that contains the basic settings for the compilation process.

Creating a Project

Set up your design as follows:

1. In the Precision RTL Synthesis software, click the **New Project** icon in the Design Bar on the left side of the Graphical User Interface (GUI). Set the **Project Name** and the **Project Folder**. The implementation name of the design corresponds to this project name.

2. 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.
3. Click the **Setup Design** icon in the Design Bar.
4. To specify a target device family, expand the Altera entry, and choose the target device and speed grade.
5. If desired, set a global design frequency and/or default input and output delays. This will constrain 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. All timing constraints are forward-annotated to the Quartus II software using Tcl scripts.

If you need to generate additional netlist files (e.g., an HDL netlist for simulation), choose **Set Options > Output > Additional Output Netlist** (Tools menu). A separate file is generated for each type that is selected (EDIF, Verilog HDL, VHDL).

Compiling the Design

To compile the design into a technology-independent implementation, click the **Compile** icon in the Design Bar.

Setting Constraints

The next steps involve setting constraints and mapping the design to technology-specific cells. By default, the Precision RTL Synthesis software maps the design 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 a static timing analysis to determine the location of the critical timing paths. Since the Precision RTL Synthesis software is fully constraint-driven, set as many constraints as possible to get the best results. Constraints include timing constraints, mapping constraints, false paths constraints, multicycle paths constraints, and constraints 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. You can create this file with a text editor or use the Precision RTL Synthesis software to generate one for you automatically on the first synthesis run. To create an initial constraint file manually, set constraints on design objects (such as clocks, design blocks, or pins) in the Design Hierarchy pane. By default,

the Precision RTL Synthesis software saves all timing constraints and attributes specified in two files: **precision_rtl.sdc** and **precision_tech.sdc**. The **precision_rtl.sdc** file contains constraints set on the RTL-level database (after compile) and **precision_tech.sdc** file contains constraints set on the gate-level database (after synthesize) 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.

 Some constraints that rarely change can also be added directly to the HDL source files by using HDL attributes or pragmas.

 For more details and examples, see the *Attributes* chapter in the *Precision Synthesis Reference Manual* at www.mentor.com.

Setting Timing Constraints

Timing constraints, based on the industry standard SDC format, are necessary for the Precision RTL Synthesis software to deliver correct results. Missing timing constraints result in incomplete timing analysis and may allow timing errors to go undetected. Precision RTL Synthesis software provides constraint analysis prior to synthesis to ensure that designs are fully and accurately constrained. All timing constraints are forward-annotated to the Quartus II software using Tcl scripts.

 Because the SDC format requires that timing constraints must be set relative to defined clocks, you must specify your clocks before applying any other timing constraints.

 For details on the syntax of SDC commands, see the *Precision RTL Synthesis Users Manual* and the *Precision Synthesis Reference Manual* available on the Mentor Graphics web site at www.mentor.com.

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 & 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. 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 `.sdc` constraint file to specify pin number constraints, I/O standards, drive strengths, and slew-rate settings.

The entries in the `.sdc` file should be in the formats shown below. For pin numbers:

```
set_attribute -name PIN_NUMBER -value "<pin number>" -port <port name>
```

For I/O standards:

```
set_attribute -name IOSTANDARDS -value "<I/O Standard>" -port <port name>
```

For drive strength settings:

```
set_attribute -name DRIVE -value "<Drive strength in mA>" -port <port name>
```

For slew rate settings:

```
set_attribute -name SLOWSLEW -value "TRUE | FALSE" -port <port name>
```

You can also set these options in the GUI. To set a pin number or other I/O setting in the Precision RTL Synthesis GUI:

1. After compiling the design, expand the **Ports** entry in the Design Hierarchy pane.
2. Expand the **Inputs** or **Outputs** entry under **Ports**.
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). Select the I/O standard from the IO_STANDARD list. For output pins, you can also select a drive strength setting and slew rate setting using the **DRIVE** and **SLOWSLEW** lists.



You can also assign pin numbers by right-clicking the pin in the Schematic Viewer.

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 in your design, based on 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. (See “[Disabling I/O Pad Insertion](#)” for more information.) To force an I/O register into the device's IOE using the GUI, perform the following steps:

1. After compiling the design, expand the **Ports** entry in the Design Hierarchy pane.
2. Under **Ports**, expand the **Inputs** or **Outputs** entry, as desired.
3. Under **Inputs** or **Outputs**, right-click the desired pin name and select **Force Register into IO**.



You can also make the assignment by right-clicking on the pin in the Schematic Viewer.

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 used) 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 gives the Precision RTL Synthesis software 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 may not be primary inputs or outputs of the chip 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. Choose **Set Options** (Tools menu).
2. On the **Optimization** page of the **Options** dialog box, turn off **Add IO Pads**, then click **Apply**.

This procedure adds the `setup_design -addio=false` command to the project file.

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:

1. After compiling the design, in the Design Hierarchy pane, expand the **Ports** entry by clicking the +.
2. Under **Ports**, expand the **Inputs or Outputs** entry.
3. Under **Inputs or Outputs**, right-click the desired pin name and select **Set Input Constraints** (right button pop-up menu).
4. In the **Port Constraints** dialog box for the selected pin name, turn off **Insert Pad**.



You can also make the assignment by right-clicking on the pin in the Schematic Viewer or by attaching a `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 potentially cause significant delays on wires and/or make a net unrouteable. On a critical path, high fan-out nets can cause delays in a single net segment and cause the timing constraints to fail. 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.

Synthesizing the Design & Evaluating the Results

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.

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 `<project name>_impl_1` naming convention.

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 areas 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 viewers allow you to easily make further constraints if needed to optimize the design.

Obtaining Accurate Logic Utilization & 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. The Quartus II software has advanced algorithms to take advantage of these FPGA 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 tools 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

After synthesis, 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, in the form of Tcl scripts. The Project Configuration script (*<project name>.tcl*) can be used 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 software, and makes timing assignments. For many devices to be compiled in the Quartus II software version 4.1 and later, the Project Configuration script calls the Place and Route Constraints script to make your timing constraints. The Place and Route Constraints script (*<project name>_pnr_constraints.tcl*) forward-annotates all timing constraints that you made in the Precision software, including false path assignments, multi-cycle assignments, timing groups, and related clocks. This integration means that you only need to enter these constraints once in the Precision software, and they can be passed automatically to the Quartus II software.

Precision RTL Synthesis also has a built-in place-and-route environment that allows you 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 Quartus II options from the **Quartus II** pages of the **Set Options** dialog box (Tools menu). On the **Integrated Place and Route** page, specify the path to the Quartus II executables in the **Path to Quartus II installation** box.

To automate the place-and-route process, click the **Run Quartus** icon in the **Quartus II** pane 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 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 Synthesis versions 2004a and later support the execution of individual Quartus II executables, such as analysis and synthesis (`quartus_map`), Fitter (`quartus_fit`), and Timing Analyzer (`quartus_tan`), for improved runtime and memory utilization during place and route. This flow is referred to as the “Quartus II Modular” flow option in Precision Synthesis and is compatible with Quartus II version 4.0 and later. By default, Precision generates a modular Quartus II Project

Configuration File (Tcl file) for Stratix II, Stratix, Stratix GX, MAX II, and Cyclone device families. In addition, when using this flow, all timing constraints that you set during synthesis are exported to the Quartus II PNR Constraints File (*<project name>_pnr_constraints.tcl*).

For other device families, Precision uses the “Quartus II” flow option, which enables the Quartus II compilation flow that existed in Precision versions earlier than 2004, and was supported in Quartus II versions earlier than version 4.0. The Quartus II Project Configuration File (Tcl file) written when using the “Quartus II” flow 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 constraint cannot take full advantage of the features in Quartus II software version 4.0 and later.

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 name>"
```

Depending on the device family, you may use one of the following flow options in the command mentioned 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 only use the “Quartus II Modular” flow. For the Stratix device family you may set 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 open the project using the generated Quartus II project file. You can view reports, run analysis tools, set options, and invoke the various processing flows available in the Quartus II software.

Running the Quartus II Software Manually

You can also 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 and Tcl files are located in the same directory (they should both be located in the implementation directory by default).
2. In the Quartus II software, open the Quartus II Tcl Console by choosing **Utility Windows > Tcl Console** (View menu).
3. Type `source <path>/<project name>.tcl ↵` at the Tcl Console command prompt.
4. Open the new project by choosing **Open Project** (File menu), browsing to the project name, and clicking **Open**.
5. Compile the project in the Quartus II software.

Megafunctions & Architecture-Specific Features



Altera provides parameterizable megafunctions including the 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 inferring them from generic HDL code.

For more details on specific Altera megafunctions, see the Quartus II Help. For more information on IP functions, consult the appropriate IP documentation.

If you want to instantiate a megafunction in your HDL code, you can use the MegaWizard Plug-In Manager to parameterize the function or 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 any available megafunction for the design. The “[Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager](#)” section 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 the “[Inferring Altera Megafunctions from HDL Code](#)” section.



For a detailed discussion on instantiating versus inferring megafunctions, see the *Recommended HDL Coding Styles* chapter in Volume 1 of the *Quartus II Handbook*. This chapter also provides details on 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. This file instantiates the megafunction (a black-box methodology) or, for some megafunctions, generates a fully synthesizable netlist for improved results using EDA synthesis tools such as Precision RTL Synthesis (a clear-box methodology).

Clear Box Methodology

Using the MegaWizard Plug-In Manager-generated fully synthesizable netlist is referred to as a clear-box methodology because the Precision RTL Synthesis software can “see” into the megafunction file. The clear box feature enables the synthesis tool to report more accurate resource utilization and timing estimates, taking better advantage of timing driven optimization.

This clear box feature of the MegaWizard Plug-In Manager can be turned on by choosing the **Generate clear box body (for EDA tools only)** in the **MegaWizard Plug-In Manager** (Tools menu) for certain megafunctions. If the 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 the “[Black Box Methodology](#)” section.

Using MegaWizard-Generated Verilog HDL Files for Clear Box Megafunction Instantiation

The MegaWizard Plug-In Manager generates a Verilog HDL instantiation template file `<output>.inst.v` for use in your Precision RTL Synthesis 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 Precision RTL Synthesis project and the information gets passed to the Quartus II software in the Precision RTL Synthesis-generated EDIF output file.

Using MegaWizard-Generated VHDL Files for Clear 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 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 Precision RTL Synthesis project and the information gets passed to the Quartus II software in the Precision RTL Synthesis-generated EDIF output file.

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 Precision RTL Synthesis 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 so does not take full advantage of the synthesis tool's timing driven optimization.

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. Do not include the megafunction variation wrapper file in your Precision RTL Synthesis project, but add it, along with your Precision RTL Synthesis-generated EDIF netlist to your Quartus II project. 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.

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. Do not include the megafunction variation wrapper file in your Precision RTL Synthesis project, but add it, along with your Precision RTL Synthesis-generated EDIF netlist to your Quartus II project.

Inferring Altera Megafunctions from HDL Code

The Precision RTL Synthesis engine automatically recognizes certain types of HDL code and maps arithmetic and relational operators, counters, 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, see the *Recommended HDL Coding Styles* chapter in Volume 1 of the *Quartus II Handbook*.

Multipliers

The Precision RTL Synthesis software detects multipliers in HDL code and infers an `1pm_mult` megafunction. The Precision RTL Synthesis software also allows you to control the device resources used to implement individual multipliers, as described below.

Controlling DSP Block Inference for Multipliers

By default, the Precision RTL Synthesis software uses DSP blocks available in Stratix-based families to implement multipliers. The default setting is **AUTO**, to allow Precision RTL the flexibility to choose logic (LUTs) and DSP blocks depending on the size of the multiplier. You can use the Precision RTL Synthesis GUI or HDL attributes to redirect the mapping to only logic elements or only DSP blocks. The options for multiplier mapping in the Precision RTL Synthesis Software are shown in [Table 9–2](#).

Table 9–2. Options for DEDICATED_MULT Parameter to Control Multiplier Implementation in Precision RTL Synthesis

Value	Description
ON	Only use the DSP blocks for implementation of multipliers, regardless of the size of the multiplier
OFF	Only use logic (LUTs) to implement multipliers
AUTO	Use logic (LUTs) and DSP blocks for implementation of multipliers depending on the size of the multipliers

Using the GUI

Perform the following steps to set the **Use Dedicated Multiplier** option in the Precision RTL Synthesis GUI:

1. Compile the design.
2. In the Design Hierarchy pane right-click the operator (**Instances > Operators**) for the desired multiplier and choose **Use Dedicated Multiplier** (right button pop-up menu).

Using Attributes

Use the `dedicated_mult` attribute to control the implementation of a multiplier in your HDL code as shown below, using the appropriate value from [Table 9–2](#):

Verilog HDL:

```
//synthesis attribute <signal name> dedicated_mult <value>
```

VHDL:

```
ATTRIBUTE dedicated_mult: STRING;
ATTRIBUTE dedicated_mult OF <signal name>: SIGNAL IS <value>;
```

The `dedicated_mult` attribute only works with 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 `dedicated_mult` attribute is set may get synthesized away by the Precision RTL Synthesis software due to 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 below:

Verilog HDL:

```
//synthesis attribute <signal name> preserve_signal TRUE
```

VHDL:

```
ATTRIBUTE preserve_signal: BOOLEAN;
ATTRIBUTE preserve_signal OF <signal name>: SIGNAL IS TRUE;
```

The following 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.

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
```

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 & 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. The software then places these functions in DSP blocks.

 The Precision RTL Synthesis software only 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. See the “[Controlling DSP Block Inference](#)” section for more information.



For more information on DSP blocks in Altera devices, see the appropriate Altera device family handbook and device-specific documentation. For details on which functions a given DSP block can implement, see the DSP Solutions Center on the Altera web site.

For more information on inferring Multiply-Accumulator and Multiply-Adder megafunctions in HDL code, see the *Recommended HDL Coding Styles* chapter in Volume 1 of the *Quartus II Handbook*.

Controlling DSP Block Inference

By default the Precision RTL Synthesis software infers the `altnmult_add` or `altnmult_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 `altnmult_add` or `altnmult_accum` megafunction in a certain module or entity. The options for this attribute are shown in [Table 9–3](#).

Table 9–3. Options for EXTRACT_MAC Attribute Controlling DSP Implementation

Value	Description
TRUE	The <code>altnmult_add</code> or <code>altnmult_accum</code> megafunction is inferred
FALSE	The <code>altnmult_add</code> or <code>altnmult_accum</code> megafunction is not inferred

To control inference, use the `extract_mac` attribute in your HDL code as shown below, using the appropriate value from [Table 9–3](#).

Verilog HDL:

```
//synthesis attribute <module name> extract_mac <value>
```

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 the “[Controlling DSP Block Inference](#)” section. See that section for syntax details.

The examples below 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.

Use of dedicated_mult and preserve_signal in Verilog HDL

```
module unsig_almult_accum1 (dataout, dataaa, datab, clk, aclr, clken);
    input [7:0] dataaa, datab;
    input clk, aclr, clken;

    output      [31:0] dataout;
    reg         [31:0] dataout;

    wire        [15:0] multa;
    wire        [31:0] adder_out;

    assign multa = dataaa * 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_almult_accum1 extract_mac FALSE
endmodule
```

Use of 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 & 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 on inferring RAM and ROM megafunctions in HDL code, see the *Recommended HDL Coding Styles* chapter in Volume 1 of the *Quartus II Handbook*.

Block-Based Design with the Quartus II LogicLock Methodology

As designs become more complex and designers work in teams, a block-based hierarchical design flow is often an effective design approach. In this approach, you perform optimization on individual sub-blocks and each sub-block may have its own output netlist file. After you optimize all of the sub-blocks, you integrate them into a final design and optimize it at the top level. You can use the LogicLock design methodology in the Quartus II software to perform block-based or team-based compilation.

The Precision RTL Synthesis software allows you to write an EDIF netlist file in which certain hierarchical blocks are optimized separately from all the others. Alternately, you can create different netlist files for different sections of a design hierarchy to make each section independent of the others. In either case, when synthesizing the entire project, only portions of a design that have been updated are changed when you compile the design. You can make changes, optimize, and re-synthesize your section of a design without affecting other sections.

Using the LogicLock design methodology, you can place each block's logic into a fixed or floating region in an Altera device. You then have the opportunity to maintain the placement and performance of your blocks in the Altera device. When you use the single netlist generated from the Precision RTL Synthesis software (or you have multiple EDIFs and all the netlists are contained in one Quartus II project), you can take advantage of the LogicLock flow to back-annotate the logic in the other regions. In this case, when you recompile with a change in one design block, the placement and assignments for unchanged blocks assigned to different LogicLock regions are not affected. Therefore, one designer can make changes to a piece of code that exists in an independent block and not interfere with another designer's changes, even if all the blocks are integrated in a top-level design. With the LogicLock design methodology, separate pieces of a design can evolve from development to testing without affecting other areas of a design.



For more information on using the LogicLock feature in the Quartus II software, see the *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*.

Hierarchy & Design Considerations

To ensure the proper functioning of the synthesis tool, you can only create separate blocks for modules, entities, or existing 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 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-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.

Creating a Design with Separate Blocks for the LogicLock Methodology

The first step in a hierarchical design flow is to ensure that different parts of your design will not affect the node names for other parts of this design. Doing so enables you to take advantage of the LogicLock incremental design flow in the Quartus II software.

You can separate your design blocks either by setting a LogicLock attribute in the Precision RTL Synthesis software, or by manually black-boxing each block that you want to be part of a LogicLock region and creating separate sub-design blocks in its individual implementation folder within the same project. This will allow you to switch back and forth between each sub-block and the top-level design without having to leave the current project file.

By setting a LogicLock attribute on certain blocks in the Precision RTL Synthesis software, you maintain separate design blocks from one easy-to-manage top-level synthesis project and only generate one EDIF netlist. Using the manual black-boxing method, you have multiple synthesis projects that may be required for certain team-based or bottom-up designs where a single top-level project is not desired.

After you create multiple EDIF netlists, you need to create the appropriate Quartus II project(s) to place and route the design.

Creating a Design with Separate Blocks Using the LogicLock Attribute in a Single Precision Project

Perform the following steps to set the **LogicLock** option in the Precision RTL Synthesis GUI to separate your design blocks and create LogicLock regions:

1. Compile the design.
2. In the **Design Hierarchy** pane, right-click a block for which you want to generate a LogicLock region (**Instances > Blocks**) and select **LogicLock** (right button pop-up menu).

3. In the **Set Attribute** dialog box, enter the name for your LogicLock region.

A Tcl command in the *<top-level>.tcl* file written by the Precision RTL Synthesis software assigns the selected block to a region of Auto size and a Floating location.

When you import the EDIF file to the Quartus II software, the specified design blocks are placed in different LogicLock regions.

Creating a Quartus II Project for an EDIF File Including LogicLock Regions

During synthesis, the Precision RTL Synthesis software creates a *<top-level>.tcl* file that provides the Quartus II software with the appropriate LogicLock assignments, creating a region for each specified design block along with the information to set up a Quartus II project.

The following is an example of the Tcl file containing the commands for the LogicLock region name “taps-region” in the module called “taps.”

```
set_global_assignment -section_id{taps_region} -name{LL_AUTO_SIZE}{NO}
set_global_assignment -section_id{taps_region} -name{LL_STATE}{FLOATING}
set_instances_assignment -section_id{taps_region} -to{|taps:u1} -name{LL_MEMBER_OF}
{taps_region}
```

These commands create a LogicLock region called `taps_region` for the `taps` design block 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.

To import the EDIF file into the Quartus II software, use the *<top-level>.tcl* file that contains the Precision RTL Synthesis assignments for all blocks in the project. This method allows the top-level designer to import all the blocks into one Quartus II project for an incremental flow. You can optimize all modules in the project at the same time.

Use the steps below to create the Quartus II project, import the appropriate LogicLock assignments, and compile the design:

1. Ensure that the EDIF and Tcl files are located in the same directory (they should both be located in the implementation directory by default).
2. In the Quartus II software, open the Quartus II Tcl Console by choosing **Utility Windows** (View menu).

3. Type `source <path>/<project name>.tcl` at the Tcl Console command prompt.



For more information on LogicLock assignments, see the *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*.

Generating a Design with Multiple EDIF Files Using Black Boxes

This section describes how to manually generate multiple EDIF files using a black-boxing technique. You can use this technique in team-based design flows when you want to synthesize each block separately in the Precision RTL Synthesis software and optimize each block separately in the Quartus II software.

Manually Creating Multiple EDIF Files Using Black-Boxes

To create multiple EDIF files manually in the Precision RTL Synthesis 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. If you want the Precision RTL Synthesis software to write out LogicLock constraints for the different blocks, use the LogicLock feature in the top-level project.

When synthesizing the projects for lower-level modules perform the following steps:

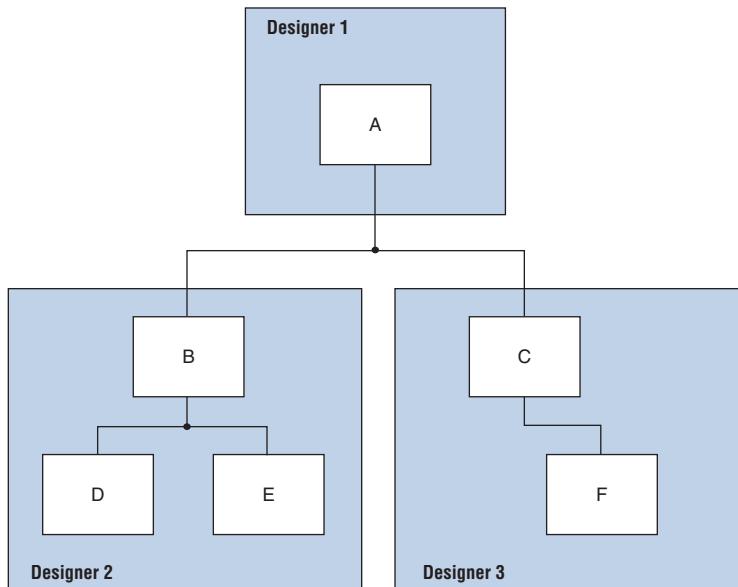
1. Turn off **Add IO Pads** on the **Optimization** page under **Set Options** (Tools menu).
2. Read the HDL files for the modules.
3. Modules may include black box instantiations of lower-level modules that are also maintained as separate EDIF files.
4. Add constraints.

When setting up the top-level design, perform the following steps:

1. Read the HDL files for top-level designs.
2. Black box lower-level modules in the top-level design.
3. Add constraints.

The sections below describe an example of black-boxing modules in a block-based and team-based design flow. [Figure 9–2](#) shows an example of a design hierarchy.

Figure 9–2. Block-Based & Team-Based Design Example



In [Figure 9–2](#), 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. One netlist is created for the top-level module A, another netlist is created for B and its submodules D and E, while a third netlist is created for C and its submodule F. To create multiple EDIF files, 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 C. Use **C.v/.vhd** and **F.v/.vhd** as the source files.
3. Generate a top-level EDIF file **A.v/.vhd** for module A. Ensure that you black-box modules B and C, which were optimized separately in the previous steps.

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, will be treated as a black box by the software. In Verilog HDL, you must provide an empty module declaration for the module that you will be treating as a black box.

A black boxing example for top-level file **A.v** follows. 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:

```
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));
    C 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 C follow here.
// These module declarations (including ports) are required for black
// boxing.

module B (data_in, clk, ld, data_out);
    input data_in, clk, ld;
    output [15:0] data_out;
endmodule

module C (d, clk, e, q);
    input [15:0] d;
    input clk, e;
    output [15:0] q;
endmodule
```

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, will be 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 boxing example for top-level file **A.vhd** follows. 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.

```
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 C 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 : C
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 outlined in this section, you will have different EDIF netlist files for each block of code. These files can now be used in the LogicLock incremental design methodology in the Quartus II software.

Creating a Quartus II Project for Multiple EDIF Files

The Precision RTL Synthesis software creates a Tcl file for each EDIF file, providing the Quartus II software with the information to set up a project. Altera recommends the following method 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 Precision RTL Synthesis software for each Precision 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 in the Quartus II software and back-annotate their blocks. 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 be back-annotated and brought into one top-level project.



For more information on creating and importing LogicLock assignments, see the *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*.

Conclusion

Advanced synthesis is an important part of the design flow. The Mentor Graphics Precision RTL Synthesis software and Quartus II design flow allow you to control how your design files will be prepared 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 optimize a design to achieve performance goals and save design time.

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. Advanced synthesis flows include the use of block-based hierarchical design methodologies.

To maximize the benefits of the LogicLock™ block-based design methodology in the Quartus® II software, you can partition a new design into a hierarchy of EDIF files during synthesis in the Synopsys FPGA Compiler II software.

This chapter describes how to automate the creation of multiple EDIF netlist files for a given hierarchy using the Synopsys FPGA Compiler II software's block-level incremental synthesis (BLIS) feature.



For more information, see the *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*.



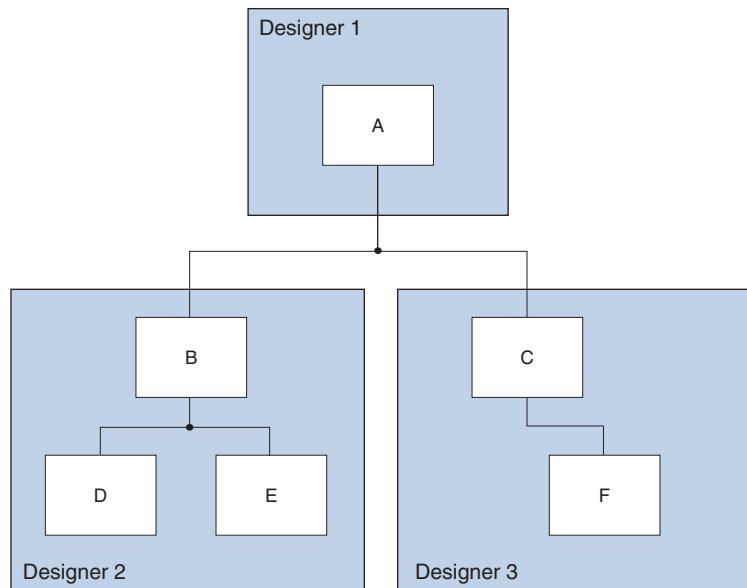
This chapter assumes that you have set-up and licensed, and are familiar with the FPGA Compiler II software.



To obtain the FPGA Compiler II software and the instructions on general product usage, go to the Synopsys web site at www.synopsys.com.

Design Hierarchy

Different modules can be defined in different files, and instantiated in a top-level file. For larger designs, like those used for Stratix® II devices, many designers can work on different modules of a design at the same time. [Figure 10–1](#) shows an example of a design hierarchy.

Figure 10–1. Design Hierarchy for Block-Based Designs

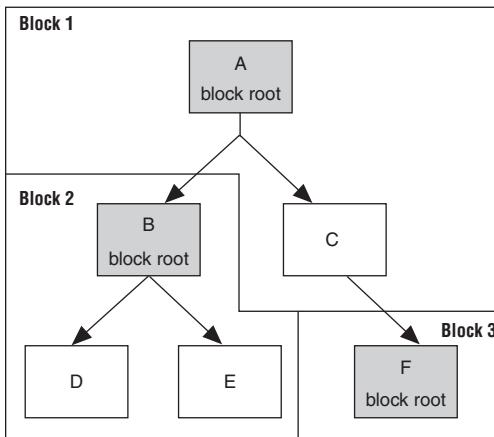
In [Figure 10–1](#), the top level of a design (A) can be assigned to one engineer (designer 1), while two engineers work on the lower levels of the design. Designer 2 works on B and its submodules (D and E) while designer 3 works on C and its submodule (F).

Block-Level Incremental Synthesis

The BLIS feature, provided with the Synopsys FPGA Compiler II software, manages a design hierarchy for incremental synthesis. The BLIS feature allows different netlist files to be created for different sections of a design hierarchy. It also ensures that only those sections of a design that have been updated will be re-synthesized when the design is compiled, reducing synthesis run time. You can change and re-synthesize a section of a design without affecting other sections of a design. The BLIS feature utilizes design units called blocks to create this functionality.

FPGA Compiler II Design Block

A block is a module or a group of modules used for incremental synthesis. Each block will have its own netlist file after synthesis. A block can be a Verilog HDL module, a VHDL entity, an EDIF netlist file, or a combination of the three. To combine these modules into a block, they should form a single tree in the design. [Figure 10–2](#) shows a block design hierarchy.

Figure 10–2. Blocks & Block Roots in a Design Hierarchy

In Figure 10–2, sections B, D, and E can be in a single block because they form a tree. Sections A, D, and F cannot form a block because they are not on the same branch of the hierarchy.

FPGA Compiler II & Quartus II Synthesis

Using the BLIS feature in the FPGA Compiler II software, you can resynthesize a netlist file for each block independently. Using the LogicLock design capability, each block's netlist file can be placed into a region on an Altera® device. The region may be stationary or floating. You can maintain the performance and placement of a block if the region is not back annotated. If a region is fixed, the placement of that portion of the design will remain the same even if other parts of a design are added.



For more information, see the *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*.

Block Root

A block root is the top module (or level) in a block's hierarchy. In Figure 10–2, A, B, and F are block roots. When a block root is declared, every module, entity, or netlist file below the block root in the hierarchy becomes part of the same block. If a new block root is declared below an existing block root, then a new block is formed. For example, in Figure 10–2, A is a block root. A is above B, C, D, E, and F in the design

hierarchy, but only C is a part of A's block. B and F were declared block roots and have formed new blocks in the design hierarchy. [Table 10–1](#) summarizes the structure of [Figure 10–2](#).

<i>Table 10–1. Synthesis in Block-Level Methodology</i>			
Block	Block Root	Member Elements	Netlist Filename
block 1	A	A, C	A.edf
block 2	B	B, D, E	B.edf
block 3	F	F	F.edf

For each defined block in the FPGA Compiler II software, a separate optimized netlist file will be created. The name of the new netlist file for each block is the same as the module, entity, or netlist file that is declared as the block's root. For example, the block root of block 1 is A. Therefore, the netlist filename after block 1 is **A.edf**.

How the BLIS Feature Works with the LogicLock Feature

When code for any module or entity defined in a block changes, the entire block is resynthesized. See [Figure 10–2](#) for the following example: If C changes, block 1 (which includes both A and C) is re-synthesized. Blocks 2 and 3 (including B, D, E, and F) are not recompiled. Since each block in a design has its own netlist file, an updated netlist file is created only for block 1 resulting in a new **A.edf** file.

Each block in the FPGA Compiler II software creates an independent netlist file after synthesis, so you can control the placement of the netlist file in LogicLock regions. Each netlist file can be placed into a separate LogicLock region in the Quartus II software. If a design region changes, only the block associated with the changed region is affected. An updated netlist file will be created in the FPGA Compiler II software for the affected block only.

During place-and-route in the Quartus II software, a LogicLock region associated with the changed netlist file will be re-run through place-and-route.

You may need to remove previous back-annotated assignments for the modified block because the node names may be different in the newly synthesized version. The placement and assignments for unchanged netlist files assigned to different LogicLock regions is not affected. You can make changes to a piece of code that exists in an independent block

and not interfere with another designer's changes. With the LogicLock design methodology, separate pieces of a design can evolve from development to testing without affecting other areas of a design.

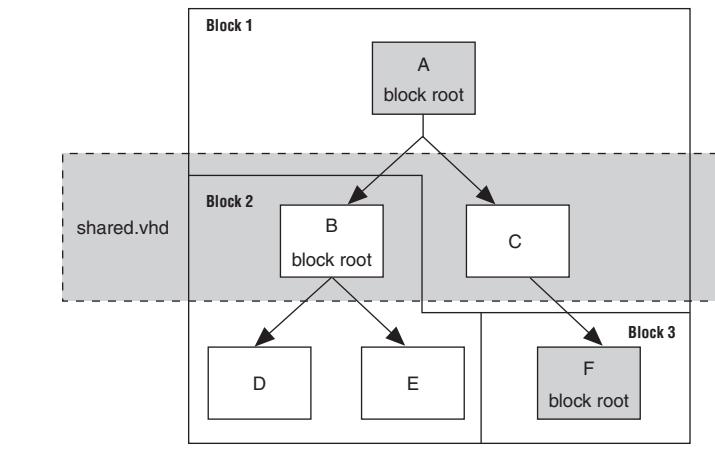
Hierarchy Considerations

You must plan your design's structure in order to use the BLIS and LogicLock features effectively. When planning a design using the BLIS and LogicLock features, keep in mind the following:

- Scope of design elements
- Organization of design elements
- Number of elements created

To ensure the proper functioning of the synthesis tool, design elements smaller than modules, entities, and netlist files cannot be declared as self-contained blocks. 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 blocks, both blocks are resynthesized when any module in the file is changed, as shown in [Figure 10–3](#).

Figure 10–3. Shared Source File Causes Re-Synthesis of Multiple Blocks



In [Figure 10–3](#), A, D, E, and F are contained in their own source files, as recommended. However, B and C share a source file, called **shared.vhd**. If C is modified in **shared.vhd**, not only are A and C updated according to the block designations in [Table 10–1](#), but B, D, and E are updated as well.

To use the BLIS feature you must ensure the following:

- Design elements defined as blocks must be modules, entities, or netlist files
- Each entity, module, or netlist file must be in its own file
- At least two blocks must be a part of the design

Time Stamp Synthesis

The resynthesis of a particular block is controlled by the time stamps of its member source files. In [Figure 10-3](#), when C is modified, the time stamp of **shared.vhd** is updated. The software sees that shared.vhd has been updated and does not know if it was module B or C that was changed, therefore, it will resynthesize blocks 1 and 2. In incremental design synthesis, only the portion of the design that was modified should be resynthesized. If you previously verified B and later made changes to C, then B is resynthesized, triggered by the updated time stamp of shared.vhd. This could change the results of any verification performed earlier on B.

When a design is planned properly using the BLIS feature, each block has a separate netlist file after synthesis, and each netlist file is updated only when its associated code is changed. This is enforced through time stamps of independent source files.

Creating & Maintaining a Design

To create and compile a project using the FPGA Compiler II software, perform the following steps:

1. Start the FPGA Compiler II software.
2. Select **New Project** (File menu).
3. Enter a project name and create a working directory.
4. Specify the source files in your design.
5. Select the top level design.
6. Select the target device (**Create Implementation** box).
7. Un-check **Skip Constraint Entry** and set the desired preferences.
8. Click **OK**. An elaborated implementation of your design appears in the **Chips** view.

Opening the Modules Constraint Table & Labeling Block Roots

To label a block root, perform the following steps:

1. Right-click on an elaborated implementation of your design (**Chips** window).
2. Select **Edit Constraints**.
3. Click the **Modules** tab.
4. Specify subdesigns as block roots in the **Block Partition** column.
5. Click **OK**.
6. Right-click on an elaborated implementation and select **Optimize Chip** to resynthesize the design.

Figure 10–4 shows how to label block roots.

Figure 10–4. Labeling Block Roots in the Edit Constraints Window

	Name	Hierarchy	Primitives	Dont Touch	Block Partition	Operator Sharing	Optimize for	Effort	Duplicate Register Merge	Location
1	<default>	Preserve	Preserve		On		Area	Low	Enable	
2	A			False	Block Root					
3	F - u1				Remove					
4	C - u2				Create a Block					
5	E - u3									
6	D - u4									
7	B - u5				Block Root					

Exporting Block-Level Netlist Files

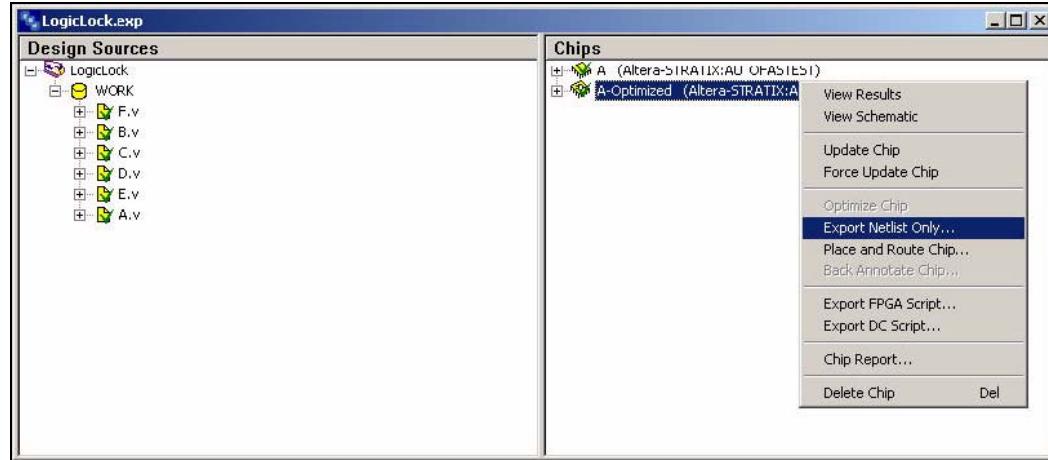
Once your design has been segmented into blocks and re-implemented, you can export netlist files. To export netlist files, perform the following steps:

1. Check that there are no red question marks over elements in the **Design Sources** or **Chips** views. The question marks indicate that a change has been made since the last update. If there are red question marks, right-click on the icons to resynthesize the design. For more information, see “[Changing Source Within a Block](#)” on page 10–8. Right-click on an optimized implementation and select **Export Netlist Only** (see Figure 10–5).

2. Click **OK** after selecting a directory for output.

One netlist file for each block is created in the directory you specified. The EDIF netlist files have the same name as their corresponding block roots.

Figure 10–5. Export Netlist File Command



Changing Source Within a Block

If you make changes to the source of a block during your design cycle, you must update your design. When you make a change to the source of a block, a red question mark will appear in the **Design Sources** window. To make a change, perform the following steps:

1. Right-click on the question mark and select **Update Chip**. Question marks will appear over the **Elaborated Implementation** and the **Optimized Implementation** icons in the **Chips** window.
2. Right-click the red question mark to update **Elaborated Implementation**.
3. Right-click the red question mark to update **Optimized Implementation**.



You must update **Elaborated Implementation** first, followed by **Optimized Implementation**.

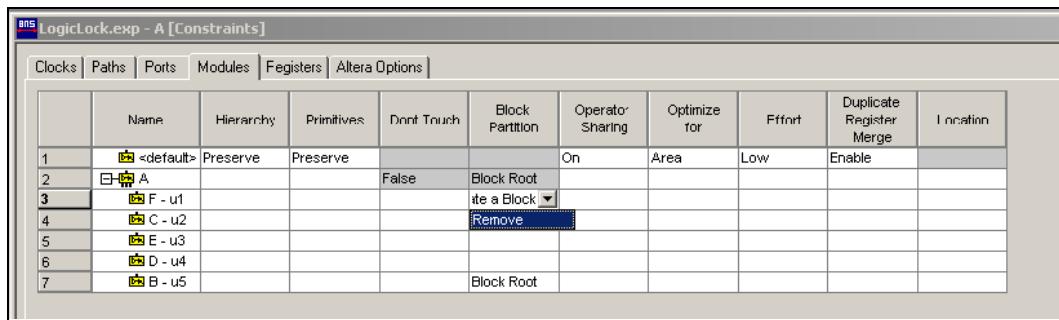
When you update all of the parts of the design, new netlist files will be created for only those parts that have been changed. You can check the time stamps of the new files to confirm this.

Removing a Block Root

If portions of your design are no longer needed, you can easily remove block roots. To remove a block root, perform the following steps:

1. Right-click on the **Elaborated Implementation (Chips** window).
2. Select **Edit Constraints**.
3. Click on the **Modules** tab and highlight the block root that you would like to remove in the **Block Partition** column.
4. Select **Remove** in the drop-down menu, see [Figure 10–6](#).

Figure 10–6. Removing a Block Root



	Name	Hierarchy	Primitives	Don't Touch	Block Partition	Operator Sharing	Optimize for	Effort	Duplicate Register Merge	Location
1	<default>	Preserve	Preserve			On	Area	Low	Enable	
2	A			False	Block Root					
3	F - u1				ite a Block					
4	C - u2				Remove					
5	E - u3									
6	D - u4									
7	B - u5				Block Root					



The top level of your design is always a block root and appears in the constraints editor. You cannot remove the block.

Using BLIS Shell Commands

You can designate block roots using the FPGA Compiler II shell with the command `set_module_block` followed by the option `true` and the path to the module, entity, or netlist file. For example, to set the module F as a block root, perform the following step:

```
fc2_shell> set_module_block true
c:\AlteraDesigns\LogicLock\F ↵
```

You can also remove a block designation using the `false` option.

```
fc2_shell> set_module_block false  
c:\AlteraDesigns\LogicLock\F ↵
```

You cannot designate a block root for top-level entities since this is the default. You also cannot designate any primitive (such as AND) as a block root because primitives are too small in scope.

Conclusion

The FPGA Compiler II software supports advanced synthesis for Altera devices and supports the LogicLock hierachal design files through the BLIS feature. The LogicLock block-based design flow uses module-based design to help you preserve performance of modules and have control over placement. Tagging modules which have separate EDIF files, you can create multiple EDIF files for use with the Quartus II software and the LogicLock block-based design feature from a single FPGA Compiler II software project.

qii51014-1.1

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.

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 DC FPGA 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 web site at the following URL:
<http://solvnet.synopsys.com/retrieve/012889.html>

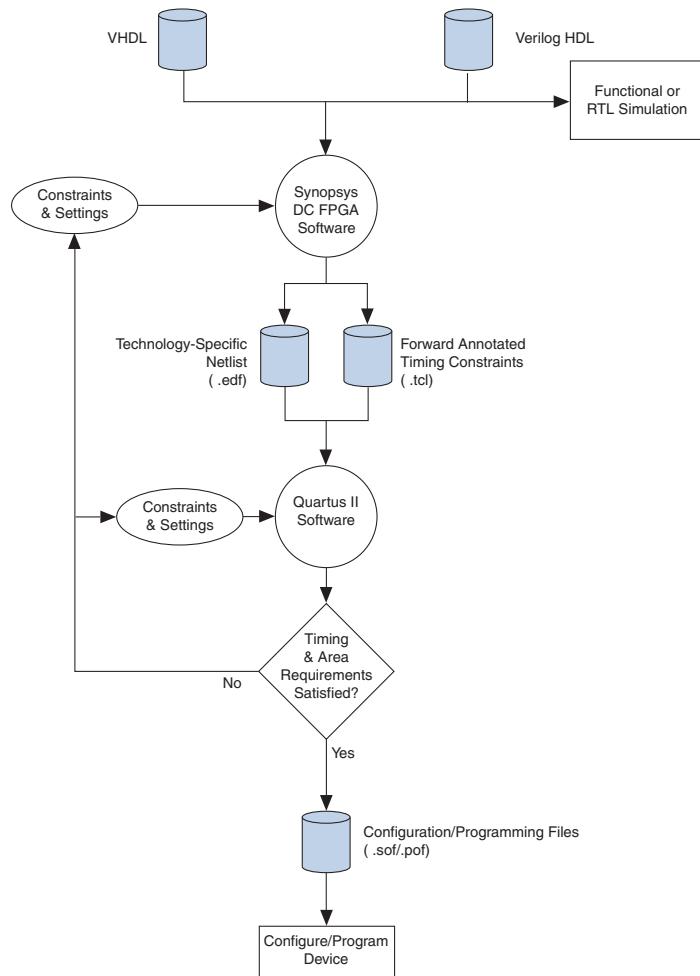
The following areas are covered in this chapter:

- 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 & 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](#).

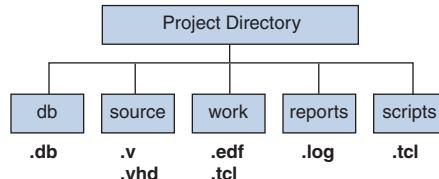
Figure 11–1. Design Flow Using the DC FPGA Software & the Quartus II Software



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 e.g. `<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 for 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.

The following example contains the recommended settings for synthesizing for the Stratix architecture:

```
# Setup file for Altera Stratix
# 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 edifout_netlist_only "true"
set edifout_power_and_ground_representation "net"
set edifout_ground_net_name "GND"
set edifout_power_net_name "VDD"
set hdlin_enable_vpp "true"
set edifout_write_properties_list "lut_function part IOSTANDARD DRIVE SLEW"
set post_compile_cost_check "false"
set_fpga_defaults altera_stratix'
```

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 & 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, see 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 11–5.



For a detailed discussion on instantiating versus inferring megafunctions, see the *Recommended HDL Coding Styles* chapter in Volume 1 of the *Quartus II Handbook*. That 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. The “[Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager](#)” section describes the MegaWizard Plug-In Manager flow with the DC FPGA synthesis software.

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.

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 create a custom megafunction variation, the MegaWizard Plug-In Manager creates either a VHDL or Verilog HDL wrapper file that instantiates the megafunction.

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 2004.06-SP1 also support altl in a non-blackbox flow (i.e. 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>.vhdl` 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 EDIF 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 EDIF 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 installment directory>/libraries/vhdl/lpm/lpm_pack.vhd`
- `<Quartus II installment 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 the `<Quartus II installment directory>/libraries/vhdl/altera_mf/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|vhdl]` along with the rest of the RTL files
2. During place-and-route in the Quartus II software, simply run the self-contained edif. You do not need to put the variation file in the edif directory.

The benefit of using the direct instantiation method is that 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 will provide optimal results.



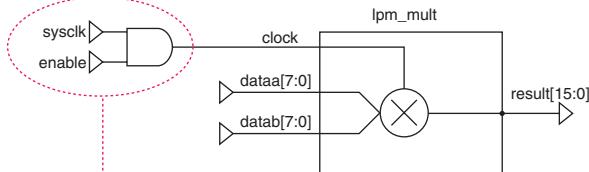
For coding style recommendations and examples for inferring megafunctions in Altera devices, see the *Recommended HDL Coding Styles* chapter in Volume 1 of the *Quartus II Handbook*.

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.

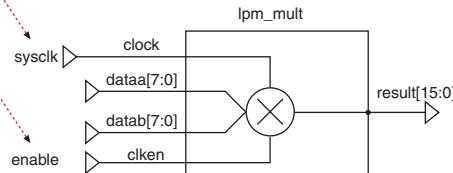
The following example shows Verilog HDL code that infers a two-port RAM that can be synthesized into an M512 RAM block of a Stratix device:

```
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

DC FPGA recognizes gated clocks and utilizes clock enable logic during synthesis where possible.



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 (Part 1 of 2)

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

Table 11-1. Supported Design File Formats (Part 2 of 2)			
Format	Description	Keyword	Extension
EDIF	Electronic design interchange format	edif	.edf

Note to Table 11-1:

- (1) The Design Compiler DB format file requires additional license keys.

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 re-use. You need to reanalyze 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 not necessary. 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 command shown below:

```
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. The usage of this command is shown here:

```
current_design <design name> ↵
```

You then need to build your design from all of the analyzed HDL files with the link command. The usage of this command is shown here:

```
link ↵
```

After linking your designs successfully in the DC FPGA software, you should specify which design you will be applying constraints to. 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 them. Use the `set_dont_touch` constraint and apply it to each module of your design that you do not want synthesized. The usage of this command is shown here:

```
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 you can specify the device for synthesis with the `set_fpga_target_device` command. The usage of this command is shown below:

```
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 for the Stratix library is shown in the following example:

```
fpga_shell-t> set_fpga_target_device -show_all
Loading db file
'./dc_fpga/libraries/fpga/altera/STRATIX/stratix.db'
Valid device names are:

Part          Pins    FFs     Speed Grades
-----
AUTO *        0       0       FASTEST
EP1S10B672   672    10570   C6 C7
EP1S10F484   484    10570   C5 C6 C7 I6
EP1S10F672   672    10570   C6 C7
EP1S10F780   780    10570   C5 C5ES C6 C6ES C7 C7ES I6
EP1S20B672   672    18460   C6 C7
EP1S20F484   484    18460   C5 C6 C7
EP1S20F672   672    18460   C6 C7 I7
EP1S20F780   780    18460   C5 C6 C7 I6
EP1S25B672   672    25660   C6 C7 I7
EP1S25F672   672    25660   C6 C6_HARDCOPY_FPGA_PROTOTYPE C7
C7_HARDCOPY_FPGA_PROTOTYPE I7
EP1S25F780   780    25660   C5 C6 C7 I6
EP1S25F1020  1020   25660   C5 C6 C7 I6
EP1S30B956   956    32470   C5 C6 C7
EP1S30F780   780    32470   C5 C5_HARDCOPY_FPGA_PROTOTYPE C6
C6_HARDCOPY_FPGA_PROTOTYPE C7 C7_HARDCOPY_FPGA_PROTOTYPE
EP1S30F1020  1020   32470   C5 C6 C7 I6
EP1S40B956   956    41250   C5 C6 C7
EP1S40F780   780    41250   C5 C5_HARDCOPY_FPGA_PROTOTYPE C6
C6_HARDCOPY_FPGA_PROTOTYPE C7 C7_HARDCOPY_FPGA_PROTOTYPE
EP1S40F1020  1020   41250   C5 C6 C7 I6
EP1S40F1508  1508   41250   C5 C6 C7
EP1S60B956   956    57120   C6 C7
EP1S60F1020  1020   57120   C6 C6_HARDCOPY_FPGA_PROTOTYPE C7
C7_HARDCOPY_FPGA_PROTOTYPE
EP1S60F1508  1508   57120   C6 C7
EP1S80B956   956    79040   C6 C7
EP1S80F1020  1020   79040   C6 C6_HARDCOPY_FPGA_PROTOTYPE C7
C7_HARDCOPY_FPGA_PROTOTYPE
EP1S80F1508  1508   79040   C6 C7

* Default part
```

For example, if you want to target the `C6_HARDCOPY_FPGA_PROTOTYPE` of the Stratix EP1S25F672 device, apply the constraint shown below:

```
set_fpga_target_device
EP1S25F672C6_HARDCOPY_FPGA_PROTOTYPE
```

Timing & 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, see Chapters 6 and 7 of the *Synopsys DC FPGA User Guide*.



For synthesis with the DC FPGA software, minimum timing analysis is not necessary, as it is primarily looking 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.

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 EDIF netlist through analysis and synthesis in the Quartus II software (`quartus_map`). All megafunctions will be expanded 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 will be 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 get better area reduction and performance improvement by ungrouping smaller blocks of logic in your design hierarchy and combining functions.

Compilation & 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 reordering 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, and saves the synthesized netlist is shown below:

```

# Setup output directories
set outdir ./design
file delete -force $outdir
file mkdir $outdir
set rptdir ./report
file delete -force $rptdir
file mkdir $rptdir
# Setup libraries
define_design_lib work-path ./$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 -f edif -hier -o $outdir/top_module.edf
write_par_constraint $outdir/top_module_quartus_setup.tcl
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 (Part 1 of 2)

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

Table 11–2. Reporting Commands (Part 2 of 2)		
Object	Command	Description
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 on the usage of these commands, see Chapter 9 of 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, VHDL, Synopsys internal DB, or EDIF.

The Quartus II software only accepts an EDIF netlist synthesized from the DC FPGA software.

Use the `write` command to save your design work. The syntax for this command is shown below:

```
write -format <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 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. The `write_par_constraint` command is explained in the following section.

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 with the following 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 your Quartus II project and compiles it. The second script is 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`.

To use this Tcl script in the Quartus II Tcl shell, type the following command at a command prompt in your project directory:

```
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
```

An example script is shown below:

```
#####
# Generated by DC FPGA 2004.06-2 on Wed Sept 15 14:21:37 2004
#
# 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
# top_module_quartus_setup.tcl
#           To execute this Tcl script in Quartus II GUI: source top_module_quartus_setup.tcl
#####

# Set the project_name variable
set project_name top_module

# Close the project if open
if [is_project_open] {
    project_close
}

# Create a new project
project_new -overwrite -family STRATIX -part AUTO $project_name

# Make global assignments
set_global_assignment -name TOP_LEVEL_ENTITY $project_name
set_global_assignment -name ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP ON
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

# 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 EDIF netlist file. The project configuration script (*<user-specified file name>.tcl*) is used to create and compile a Quartus II project containing your EDIF 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 & Route with the Quartus II Software

After you have created your Quartus II project and successfully loaded your EDIF 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 on these advance features, area optimization, and timing closure, please refer to *Quartus II Handbook*.

You can use the Quartus II software to provides 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, please refer to the *Quartus II Support for HardCopy Devices* chapter in 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.

Introduction

As FPGA designs grow in size and complexity, and as several design engineers are involved in coding and synthesizing different design blocks, the ability to analyze how your synthesis tool interprets your design becomes critical. The Quartus® II RTL Viewer and Technology Map Viewer provide powerful ways of viewing your initial and fully mapped synthesis results during the debugging, optimization, or constraint entry process.

The first sections of this chapter explain the parts of the RTL Viewer and Technology Map Viewer. The following sections describe how to navigate and filter in the schematics, probe to other windows within the Quartus II software, view a timing path from the Timing Analyzer report, and other features of the schematics. The final section “[Using the RTL & Technology Map Viewers to Analyze Design Problems](#)” on page 12-32 presents potential uses of the viewers to analyze your design and help save valuable verification time.

RTL Viewer Overview

The Quartus II RTL Viewer allows you to view a register transfer level (RTL) graphical representation of your initial Quartus II integrated synthesis results or your third-party netlist file within the Quartus II software.

You can view your Quartus II results after Analysis & Elaboration when your design uses Verilog HDL (.v), VHDL (.vhd), AHDL (.tdf), schematic block design files (.bdf) or 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.

The Quartus II RTL Viewer presents a schematic view of the design netlist after analysis and elaboration, or netlist extraction, by the Quartus II software, but before the synthesis or fitter optimization algorithms have taken place. This view is not the final structure of the design, since optimizations have not yet occurred, but it is the closest view possible to your original source design. If your design uses 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 as written by your synthesis tool.

Certain optimizations are performed on the netlist to improve readability in the viewer. Logic with no fan-out (i.e., its outputs are unconnected) and logic with no fan-in (i.e., its inputs are unconnected) are removed from the display. Internally-used TRI_BUS tri-state buffer primitives are removed and bidirectional I/O pins in the design are connected directly to their sources. Default connections such as VCC and GND are not shown. The RTL Viewer groups pins, nets or wires, module ports, and certain logic into buses where appropriate. Constant bus connections are also grouped, and values are displayed in hexadecimal format. NOT gates are converted to bubble inversion symbols in the schematic, and chains of equivalent combinational gates are merged into a single gate.

To run the RTL Viewer for a Quartus II project, you must first analyze the design by choosing **Start > Start Analysis & Elaboration** (Processing menu). You can also perform a full compilation or any process that includes the initial Analysis & Elaboration stage of the compilation flow. To run the viewer choose **RTL Viewer** (Tools menu), or select **RTL Viewer** from the **Applications** toolbar.



The **Applications** toolbar does not appear by default in the Quartus II user interface. To add the toolbar, choose **Customize** (Tools menu) and turn on the **Applications** toolbar on the **Toolbars** tab.

Technology Map Viewer Overview

The Quartus II Technology Map Viewer provides a low-level, technology-specific, graphical representation of your design after the synthesis process that includes mapping the design into the target technology. The Technology Map Viewer shows the hierarchy of atom primitives (such as device logic cells and I/O ports) in your design.



The port names of each hierarchy level are not maintained through synthesis so hierarchy instances have generic port names such as IN1, OUT1.

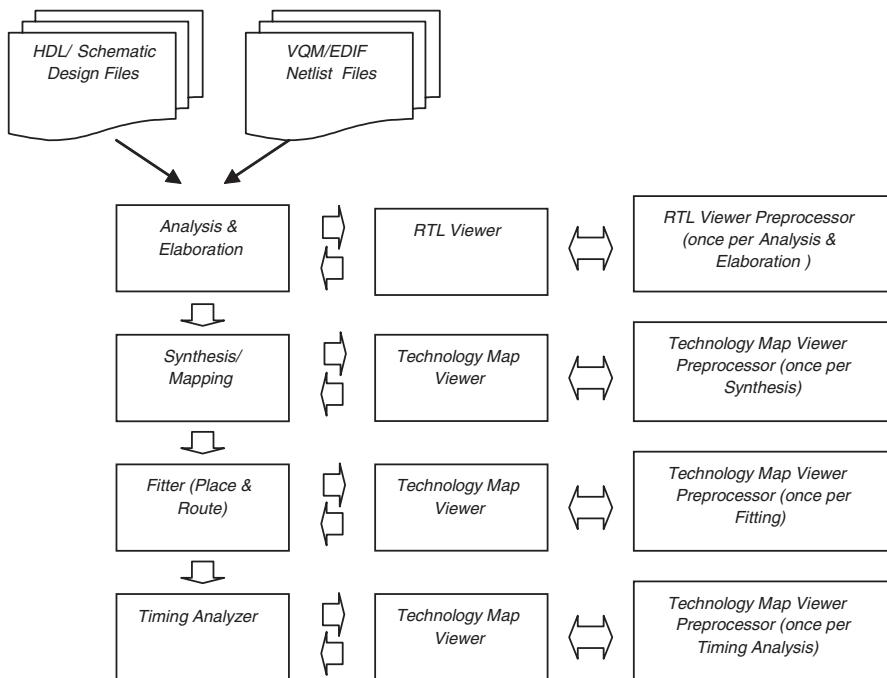
You can view your Quartus II technology-mapped results after synthesis, fitting, or timing analysis. To run the Technology Map Viewer for a Quartus II project, you must first synthesize and map the design to the target technology by choosing **Start > Start Analysis & Synthesis** (Processing menu). You can also perform a full compilation, or any process that includes the synthesis stage of the compilation flow. If you have completed the Fitter stage, the Technology Map Viewer shows any changes made to your netlist by the Fitter, such as physical synthesis optimizations. If you have completed the Timing Analysis stage, you can locate timing paths from the Timing Analyzer report in the Technology Map Viewer (see “[Viewing a Timing Path in the Technology Map Viewer](#)” on page 12–26 for details).

After the desired compilation stage is complete, you can launch the viewer by choosing **Technology Map Viewer** (Tools menu), or by selecting **Technology Map Viewer** from the **Applications** toolbar.

Quartus II Design Flow with the RTL & Technology Map Viewers

The first time you open either viewer after the appropriate compilation stage, 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 RTL Viewer and Technology Map Viewer fit into the basic Quartus II design flow.

Figure 12–1. Quartus II Design Flow Including the RTL Viewer & Technology Map Viewer



If you choose to open one of the viewers without first performing the appropriate compilation stage, the viewer does not appear. The Quartus II software issues an error instructing you to run the necessary compilation stage and restart the viewer.

The 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 view 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 and the Quartus II software issues an error when you try to open the viewer.

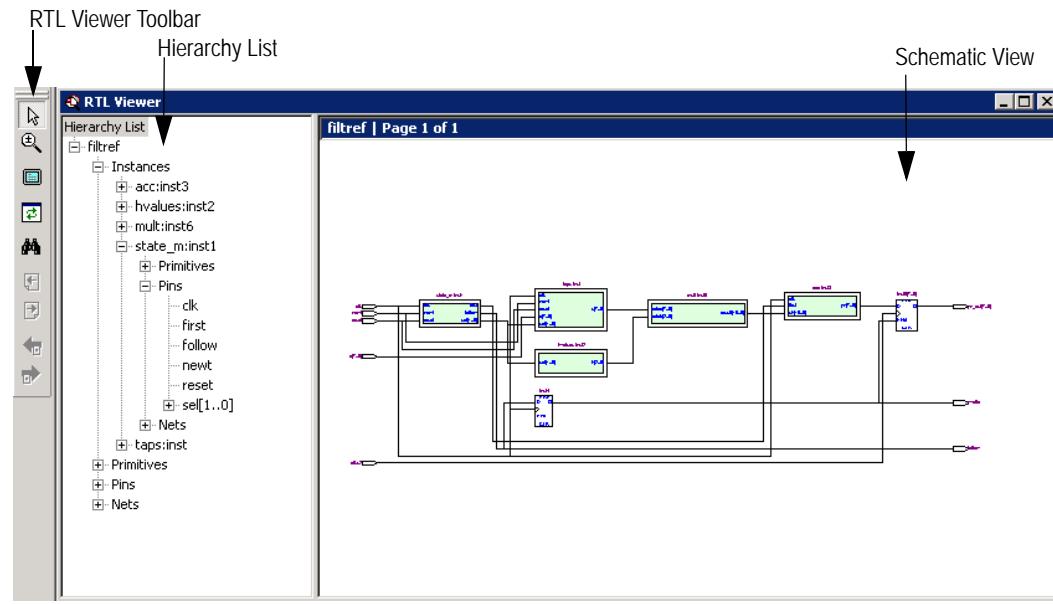


If the viewer window is open when you start a new compilation, the viewer closes automatically. You must open the viewer again to view the new design netlist.

Introduction to the User Interface

The RTL Viewer window and Technology Map Viewer window each consist of two main parts, as shown in [Figure 12-2](#) for the RTL Viewer, the schematic view, and the hierarchy list. The **RTL Viewer** or **Technology Map Viewer** toolbar also appears when you open the corresponding viewer. The toolbars provide tools for use with the schematic view.

You can only have one RTL Viewer and one Technology Map Viewer window open at a time, although each window may 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, or moved in front of or behind other windows.

Figure 12–2. RTL Viewer Window & RTL Viewer Toolbar

Schematic View

The schematic view is shown on the right-hand side of the RTL Viewer and Technology Map Viewer, and contains a schematic representing the design logic in the netlist. This is the main screen for viewing your gate-level netlist in the RTL Viewer and your technology-mapped netlist in the Technology Map Viewer.

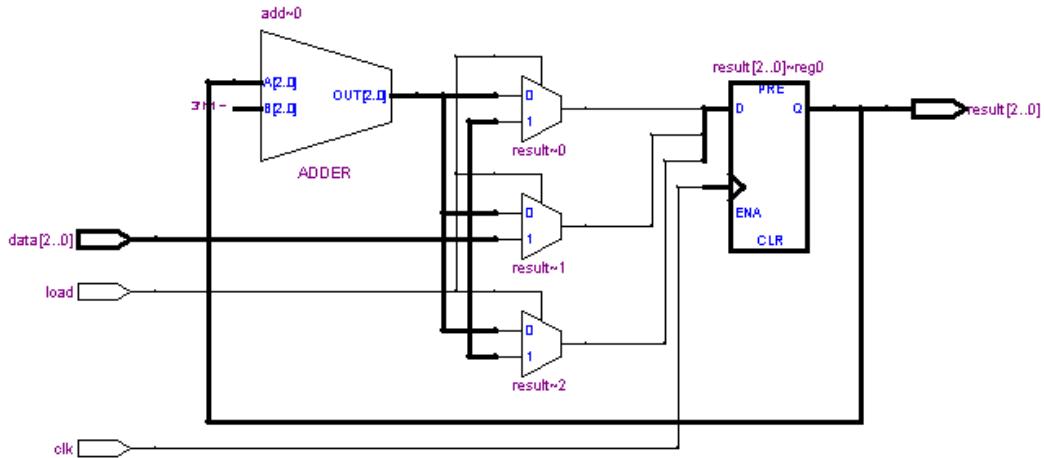
Schematic Symbols

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. The “[Code Sample for Counter Schematic Shown in Figure 12–3](#)” section shows the Verilog HDL code that was read into the Quartus II software to generate this schematic. In this example, there are multiplexers and a bus of registers (see [Table 12–1](#)) along with an ADDER operator (see [Table 12–2](#)) inferred by the counting function in the HDL code.

The schematic 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



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;
    always @ (posedge clk)
        if (load)
            result <= data;
        else
            result <= result + 1;
endmodule
```

Figure 12–4 shows a portion of the corresponding Technology Map Viewer schematic when the design is compiled targeting 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 look-up table (LUT) of the LCELL. The hexadecimal number in parentheses below each LCELL primitive represents the LUT mask, which is a hexadecimal representation of the LUT output.

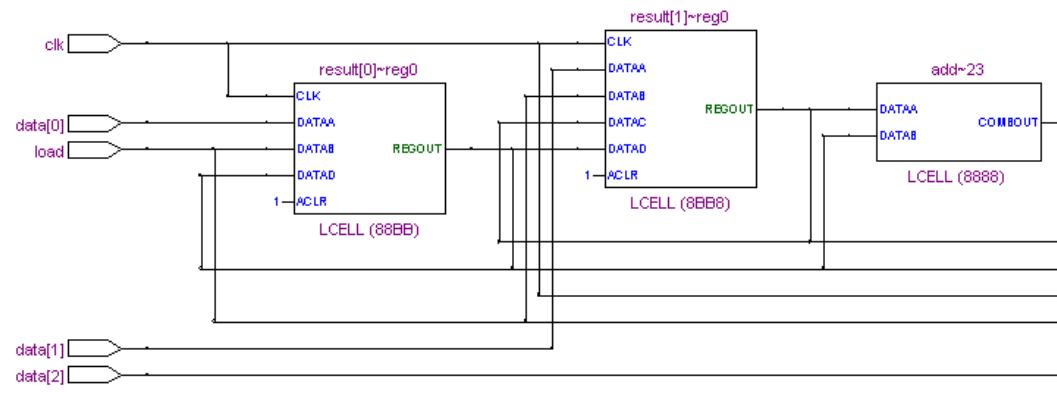
Figure 12–4. Example Schematic Diagram in the Technology Map Viewer

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.



The logic gates and operator primitives appear only in the RTL Viewer, the logic in the Technology Map Viewer is represented by atom primitives such as LCELL.

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, while 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 (see “Partitioning the Schematic into Pages” on page 12–15). To go to the page that contains the source or the destination, right-click on the net and choose the page from the right-click pop-up menu (see “Following Nets Between Schematic Pages” on page 12–17).

Table 12–1. Symbols in the Schematic View (Part 2 of 3)

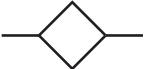
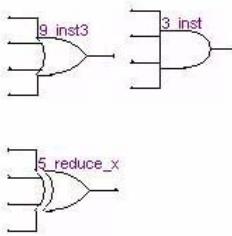
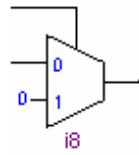
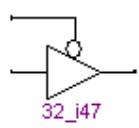
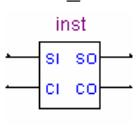
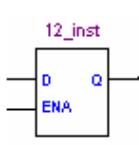
Symbol	Description
	A connector representing a port relationship between two different hierarchies. A connector indicates that a path passes through a port connector in a different level of hierarchy.
	An OR, AND, or XOR gate primitive (the number of ports can vary). A small circle (bubble symbol) on an input or output indicates that the port is inverted.
	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 (see “Operator Symbols in the Schematic View” on page 12–10).
	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.
	A CARRY_SUM buffer primitive, where SI represents SUM IN, SO represents SUM OUT, CI represents CARRY IN, and CO represents the CARRY OUT port of the buffer.
	A latch primitive with D data input, EN enable input, and Q data output.

Table 12–1. Symbols in the Schematic View (Part 3 of 3)

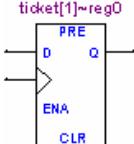
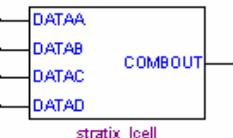
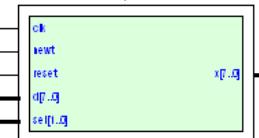
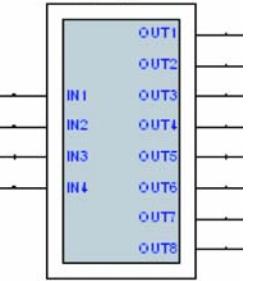
Symbol	Description
DFFE/ DFFEA/DFFES/DFFAES 	A DFFE (data flip-flop with enable) primitive, with the same ports as a latch and a clock trigger. The other flip-flop primitives are similar: DFFEA (data flip-flop with enable and asynchronous load) primitive with additional ALOAD asynchronous load and ADATA data signals, DFFES (data flip-flop with enable and synchronous load) with additional SCLR synchronous clear, SDATA data and SLOAD load ports, and DFFAES (data flip-flop with enable and both synchronous and asynchronous load) which is a combination of DFFEA and DFFES with ASDATA as the secondary data port. Preset (PRE) and Clear (CLR) connections are also shown if these ports are connected.
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 ports names, the primitive or operator type, and its name. The figure shows a stratix_lcell WYSIWYG primitive, with DATAA to DATAD and COMBOOUT port connections. This type of LCELL primitive would be found in the RTL Viewer if the source design was a VQM or EDIF netlist. The Technology Map Viewer contains similar primitives for technology-specific atom primitives.
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 coloring. The symbol displays the instance name. To open the schematic for the lower-level hierarchy by right-click and choose the appropriate command (see “Traversing the Design Hierarchy” on page 12–18).
Encrypted Instance 	A user-defined encrypted instance in the design, indicated by the double outline and gray coloring. The symbol displays the instance name. You cannot open the schematic for the lower-level hierarchy, because the source design is encrypted.

Table 12–2 lists and describes the additional higher-level operator symbols used in the RTL Viewer schematic view.

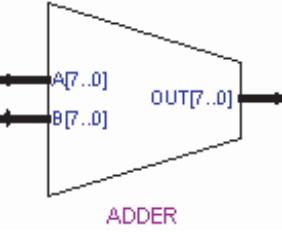
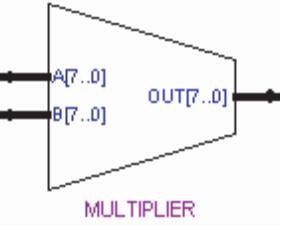
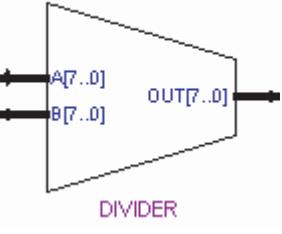
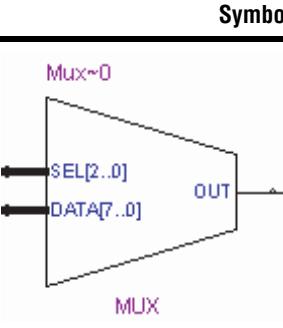
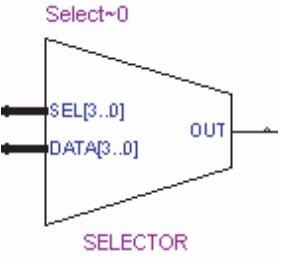
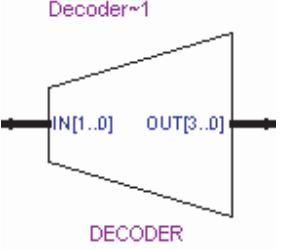
Table 12–2. Operator Symbols in the Schematic View (Part 1 of 3)	
Symbol	Description
speed.legal 	A particular state of an finite state machine, with the following ports: DATAIN - input data that control the state OUT1 - output of that state sm_clk - Clock input feeding the state sm_reset - Reset input feeding the state sm_enable - Clock Enable signal feeding the state
add~0 	An adder operator: OUT, A, and B
mult~0 	A multiplier operator: $OUT = A \times B$
div~0 	A divider operator: $OUT = A / B$

Table 12–2. Operator Symbols in the Schematic View (Part 2 of 3)

Symbol	Description
 shift_left~0 LEFT_SHIFT	A left shift operator: $OUT = (A \ll COUNT)$
 shift_right~0 RIGHT_SHIFT	A right shift operator: $OUT = (A \gg COUNT)$
 mod~0 MODULO	A modulo operator: $OUT = (A \% B)$
 LessThan~0 LESS_THAN	A less than comparator: $OUT = (A \leq B : A < B)$

Table 12–2. Operator Symbols in the Schematic View (Part 3 of 3)

Symbol	Description
 <p>Mux~0</p> <p>SEL[2..0]</p> <p>DATA[7..0]</p> <p>OUT</p> <p>MUX</p>	A multiplexer: OUT = DATA [SEL] The data range size is $2^{\text{sel range size}}$
 <p>Select~0</p> <p>SEL[3..0]</p> <p>DATA[3..0]</p> <p>OUT</p> <p>SELECTOR</p>	A multiplexer with one-hot select input (and more than two input signals).
 <p>Decoder~1</p> <p>IN[1..0]</p> <p>OUT[3..0]</p> <p>DECODER</p>	A binary number decoder: OUT = (binary_number (IN) == x) for x=0 to x= $2^{(\text{n}+1)} - 1$

Selecting an Item in the Schematic View

To select items 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 (node or wire net) in the schematic view to highlight it in red.

You can 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 or 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 select all nodes

in the area (instances, primitives, and pins), but not the nets. The **Viewer Options** dialog box provides an option to select nets as well. Right-click in the schematic and select **Viewer Options**. In the **Net Selection** section, turn on the **Select entire net when segment is selected** option.

The items selected in the schematic view are selected in the hierarchy list automatically (see the “[Hierarchy List](#)” on page 12–13). The list expands automatically if required to show the selected entry (note that the list does not collapse automatically when entries are not being used).

When you select a node or port in the schematic view, the node or port 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, nets are connected in other hierarchies of the design, but these nets are still highlighted in the current hierarchy when one of the nets is selected. If this additional highlighting is confusing and you prefer that these nets not be highlighted, the **Viewer Options** dialog box provides an option so that the net is only highlighted on the current hierarchy. Right-click in the schematic and select **Viewer Options**. In the **Net Selection** section, turn on the **Limit selections to current hierarchy** option.

Hierarchy List

The hierarchy list is displayed on the left-hand side of the viewer window, and displays the entire netlist in a “tree” format based on its hierarchical levels. Using the hierarchy list, you can traverse through the design hierarchy levels as desired and view the logic schematic for each. You can also select nodes in the list to highlight in the schematic view.

For each module in the design hierarchy, the hierarchy list displays the following entries:

- **Instances**—Modules or instances in the design that can be expanded to lower hierarchy levels.
- **Primitives**—Low-level nodes that cannot be expanded to any lower hierarchy level. These include registers and gates (in the RTL Viewer when using Quartus II integrated synthesis), or logic cell atoms (in the Technology Map Viewer or in the RTL Viewer when using a VQM or EDIF from third-party synthesis software).

- **Pins**—The I/O ports in the current level of the hierarchy.
 - The pins are device I/O pins when viewing the top hierarchy level and are I/O ports of the module when viewing the lower levels.
- When a pin represents a bus or array of pins, you can expand the pin entry in the list view to see the individual pin names.
- **Nets**—The nets or wires that connect the nodes (instances, primitives, and pins). When a net represents a bus or array of nets, you can expand the net entry in the tree view to see the individual net names.

Click the + icon to expand any of the entries.

Selecting an Item in the Hierarchy List

When you click any instance, primitive, pin, or net name in the hierarchy list, the viewer performs the following actions:

- Displays the hierarchy and page that contain the selected item in the schematic view if it is not currently displayed
- Centers the current schematic page to include the selected item, if needed
- Highlights the selected item in red in the schematic view

You can select multiple items by pressing the Shift or Ctrl key while selecting with your mouse.

Navigating the Schematic View

Zooming & Magnification

You can control the magnification of your schematic through 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** which you can enable on the **Toolbars** tab of the **Customize** dialog box (Tools menu).

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 default normal size. Select **Zoom In** to see less of the image in a larger size, and

select **Zoom Out** to see more of the image (when the entire image is not displayed) in a smaller size. The **Zoom** command allows you to specify a magnification percentage (where 100% is considered the normal size for the schematic symbols). The **Fit Selection in Window** command zooms into the selected nodes in a schematic. Use the **Selection Tool** to select one or more nodes (instances, primitives, pins, nets), then select **Fit Selection in Window** to enlarge the area covered by the selection. This feature is helpful when you have located a node of interest in a large schematic. Once a node is selected, you can easily zoom in to view that 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 on the location you clicked, and right-clicking on the schematic (or pressing the Shift key and clicking) zooms out and centers on the location you clicked. When the **Zoom Tool** is selected, you can also zoom in to a certain portion of the schematic by selecting a rectangular box area with your mouse cursor. To select a box, move your mouse to one corner of the section to be enlarged, click the mouse button, and drag the mouse to the opposite corner of the box, then release the mouse button. The schematic is enlarged to show the area you selected.

In addition, if your mouse has a wheel button, you can press the **Ctrl key** and rotate the wheel up and down to zoom in and out. The zoom focuses on the center of the schematic view if you have not made a selection in the schematic. If something in the view is selected, the zoom centers the view on the selected items.

By default, the viewers maintain the zoom level when performing filtering operations on the schematic (see “[Filtering in the Schematic View](#)” on page 12–19). To change the behavior so that the zoom level is always reset to “Fit in Window”, turn off **Maintain Zoom Level** on the RTL/Technology Map Viewer page in the **Options** dialog box (Tools menu).

Partitioning the Schematic into Pages

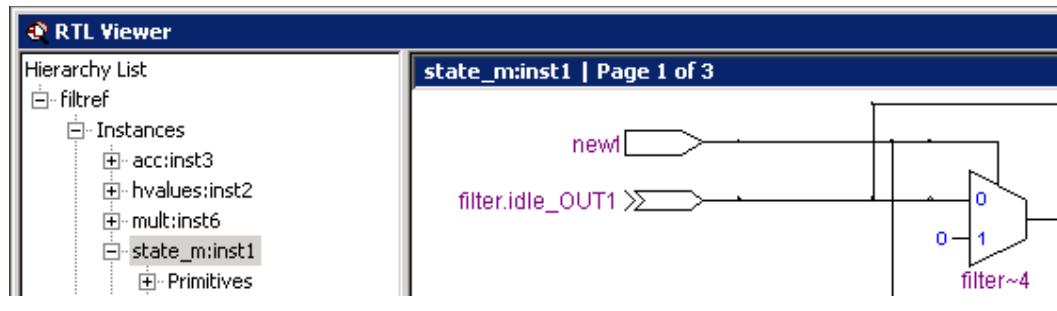
For large design hierarchies, the RTL Viewer and Technology Map Viewer partition your netlist into multiple pages in the schematic view. You can control how much of the design you would like to see on each page under **Display Settings** on the **RTL/Technology Map Viewer** tab of the **Options** dialog box (Tools menu).

The **Nodes per page** option specifies the number of nodes per partitioned page. The default value is 50 nodes, and the range is 1 to 1,000. The **Ports per page** option specifies the number of ports (or pins) per partitioned

page. The default value is 1,000 ports, and the range is 1 to 2,000. 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.)

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 (i.e., Page <current page number> of <total number of pages>). [Figure 12–5](#) shows an example.

Figure 12–5. RTL Viewer Title Bars Indicating Page Number Information



When you change the number of nodes or ports per page, it applies only to all 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 shown in [Figure 12–6](#).

Figure 12–6. Refresh Button



Moving Between Schematic Pages

You can move to another schematic page with the **Previous Page** and **Next Page** options (View menu), or by clicking **Previous Page** and **Next Page** in the viewer toolbar.

To go to a particular page of the schematic choose **Go To** (Edit menu), or right-click in the schematic view, then select the **Go To** command and select the page number in the **Page** list.

Following Nets Between Schematic Pages

Input and output connectors are used to represent nodes that connect between pages of the same hierarchy. Right-clicking on a connector displays a menu of commands that you can use to trace the net through pages of the hierarchy.

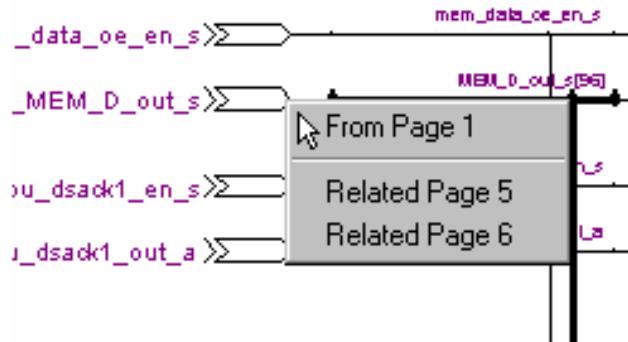


When the viewer opens a new page (after you right-click to follow a connector port), the page is centered on the particular source or destination net with the same zoom factor as the previous page. However, if you wish to trace a specific net to the new page of the hierarchy, Altera recommends that you select the desired net before you right-click to traverse between pages of the hierarchy.

Input Connectors

Figure 12–7 shows an example of the menu that appears when you right-click an input connector. The **From** command opens the page that contains the source of the signal. The **Related** commands, if applicable, open the specified page that contains another connection fed by the same source.

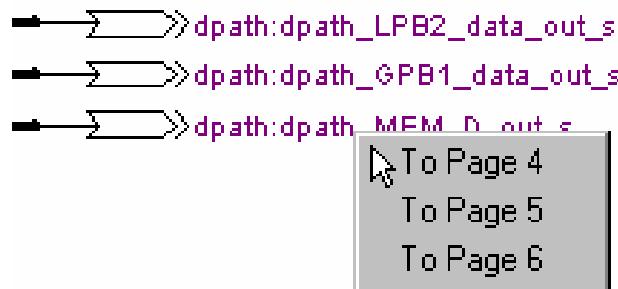
Figure 12–7. Input Connector Right Button Pop-Up Menu



Output Connectors

Figure 12–8 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 12–8. Output Connector Right Button Pop-Up Menu



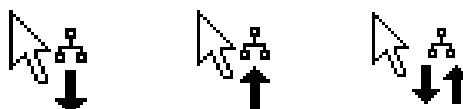
Traversing the Design Hierarchy

You can open different levels of the hierarchy in the schematic view using the hierarchy list (as described in “[Hierarchy List](#)” on page 12–13), or by using 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 that shows 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 in the hierarchy (as shown in Figure 12–9). To open the next hierarchy level, right-click in that area of the schematic and select **Hierarchy Down** or **Hierarchy Up** (right button pop-up menu), as appropriate, or double-click in that area of the schematic.

Figure 12–9. Mouse Pointers Indicating that Hierarchy Down, Hierarchy Up, or Both Down & Up are Available



Moving Back & Forward Through Schematic Pages

After changing the page view, you can go back to the previous view with the **Back** command (View menu), or by clicking **Back** in the viewer toolbar. You can return to the page view seen before going **Back** with the **Forward** command (View menu), or by clicking **Forward** in the viewer toolbar. You can only go **Forward** if you have not made any changes to the view since going **Back**.



Back and **Forward** are used to switch between page views; they do not undo an action such as a selecting a node.

Go to Net Driver

To locate the source of a particular net in the schematic view, select the net to highlight it, right-click on it, and choose one of the commands under **Go to Net Driver: Current page, Current hierarchy, or Across hierarchies**. The **Go to Net Driver > Current page** command locates the source or driver on the current page of the schematic only. The **Go to Net Driver > Current hierarchy** command locates the source within the current level of hierarchy, even if it is located on another page of the netlist schematic. The **Go to Net Driver > Across hierarchies** command locates the net's driver across hierarchies until it reaches its 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 (i.e., does not keep any filtering results).

Filtering in the Schematic View

Filtering allows you to filter out nodes and nets in your netlist to view only a logic path of interest.

You can filter your netlist by selecting the nodes, ports of a node, or nets that are part of the path you want to see. Right-click on a node, port, or net, then choose **Filter** and select the appropriate filter command, as described below. The viewer generates a new page showing the netlist that remains after filtering. You can go back to the netlist page before it was filtered using the **Back** command (described in “[Moving Back & Forward Through Schematic Pages](#)” on page 12-19).



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.

The viewers offer the following filtering commands: **Sources**, **Destinations**, **Sources & Destinations**, **Between Selected Nodes**, and **Selected Nodes and Nets**.

- **Filter > Sources**—This command filters out all but the source of the selected node, port, or net. If a node is selected, the filtered page shows all the sources of this node's input ports. See [Figure 12-10](#) for an example. If an input port is selected, the filtered page shows only the input source nodes that feed this port. If you select an output port of a node and filter by source, the filtered page shows only the selected node. If a net is selected, the filtered page shows the sources that feed the net.
- **Filter > Destinations**—This command filters out all but the destinations of the selected node or port. If a node is selected, the filtered page shows all the destination of this node's output ports. See [Figure 12-10](#) for an example. If an output port is selected, the filtered page shows only the fan-out destination nodes fed by this port. If you select an input port of a node and filter by destination, the filtered page shows only the selected node. If a net is selected, the filtered page shows the destinations fed by the net.
- **Filter > Sources & Destinations**—This 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 node, port, or net. See [Figure 12-10](#) for an example.
- **Filter > Between Selected Nodes**—This command shows the nodes in the path between two or more selected nodes. (For this option, selecting a port of a node is the same as selecting the node.) See [Figure 12-11](#) for an example.
- **Filter > Selected Nodes & Nets**—This command creates a filtered page that shows only the selected node(s) and /or nets and, if applicable, the connections between those nodes and /or net(s). See [Figures 12-12](#) and [12-13](#) for an example. If a net is selected, the filtered page shows the immediate sources and destinations of the selected net.

For all the filtering commands, the viewer stops tracing through the netlist to obtain the filtered netlist when it reaches one of the following:

- A pin.
- A specified number of filtering levels (counting from the selected node or port; 3 by default).
 - Specify the **Number of filtering** levels in the **Filtering** section of the **RTL/Technology Map Viewer Options** dialog box. Right-click in the schematic and choose to open the 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; on by default).
 - Turn the **Stop filtering at register** option on or off in the **Filtering** section of the **RTL/Technology Map Viewer Options** dialog box. Right-click in the schematic and choose **Viewer Options** to open the dialog box.

Examples of Filtered Netlists

Figure 12–10 shows an example of the **Sources**, **Destinations**, and **Sources & Destinations** filtering commands for the `inst4` node highlighted in the schematic. Figure 12–11 shows an example of the **Between Selected Nodes** filtering command between the `inst2` and `inst3` nodes highlighted in the schematic. The nodes in the appropriate box are shown in the filtered page when you select the corresponding command.

Figure 12–10. Example Schematic with “Sources,” “Destinations,” & “Sources & Destinations” Filtering for `inst4`

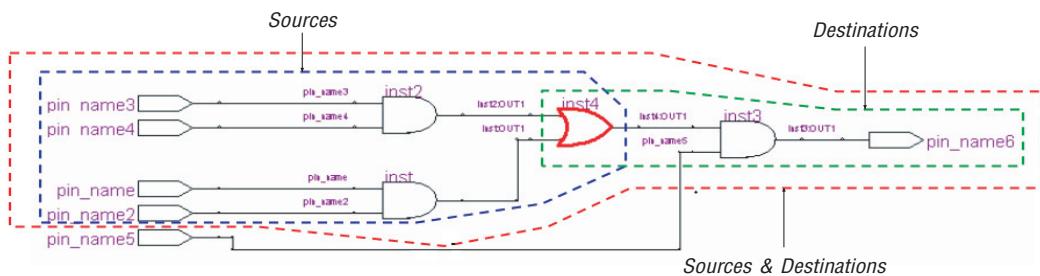
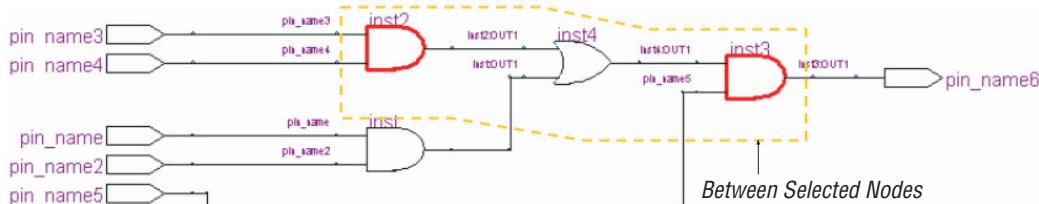
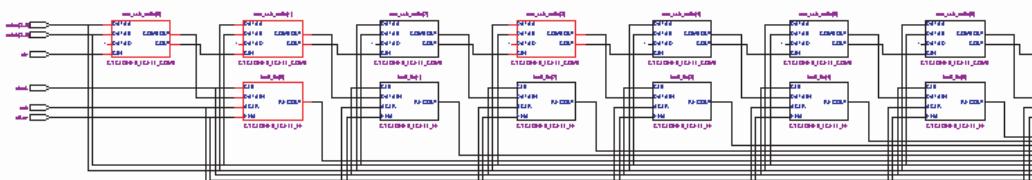
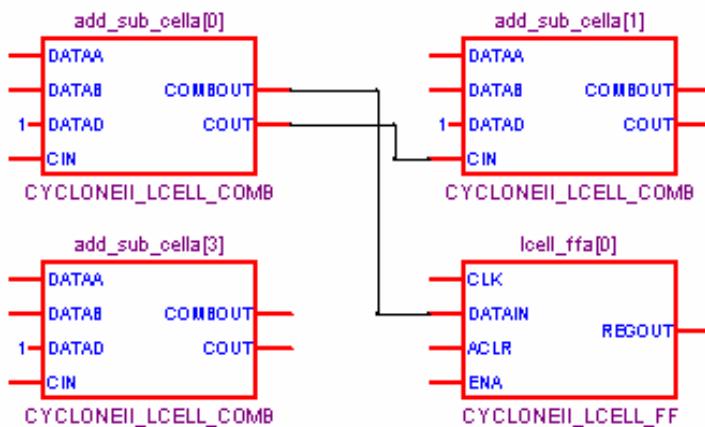


Figure 12–11. Example Schematic with “Between Selected Nodes” Filtering Between `inst2` & `inst3`



Figures 12–12 and 12–13 show an example of the **Selected Nodes & Nets** filtering command. Figure 12–12 shows several nodes highlighted in red in a schematic. Figure 12–13 shows the new schematic that is created after the **Selected Nodes & Nets** command is applied.

Figure 12–12. Example Schematic with Nodes Selected**Figure 12–13. Example Schematic from Figure 12–12 after Filtering with Selected Nodes & Nets Command**

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 hierarchical boundaries. A diamond symbol appears on the border to represent the port relationship between two different hierarchies. See [Figures 12–14](#) and [12–15](#) for examples.

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 select **Viewer Options**. In the **Filtering** section, turn off the **Filter across hierarchy** option.

If you would like to disable the box hierarchy display, turn off **Show box hierarchy** on the **RTL/Technology Map Viewer** page in the **Options** dialog box (Tools menu).



Netlists of the same hierarchy that are displayed over more than one page are not grouped with a box.

Figures 12–14 and 12–15 show examples of filtering across hierarchical boundaries. Figure 12–15 shows an example after the **Filter > Sources** command 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 12–15 shows a larger example after the **Filter > Sources** command 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 12–14. Filtering Across Hierarchical Boundaries, Small Example

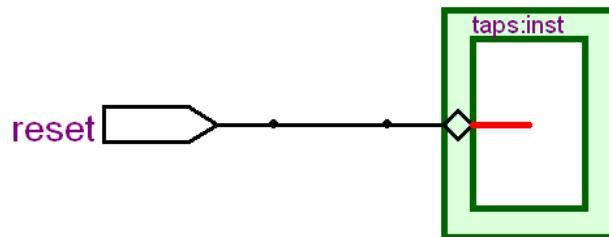
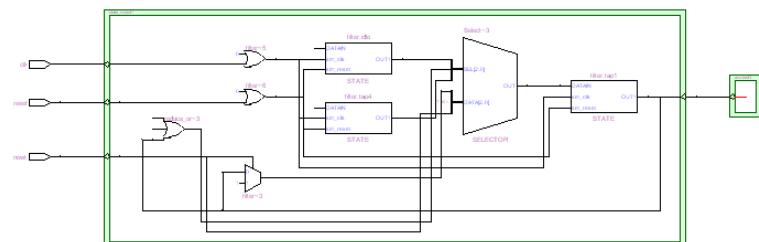


Figure 12–15. Filtering Across Hierarchical Boundaries, Large Example



Expanding a Filtered Netlist

After a netlist is filtered, there may be unconnected ports that are not part of the main path through the netlist. The two expansion features, immediate expansion and the **Expand** command, allow you to add the fan-out or fan-in signals of unconnected ports to the schematic display of a filtered netlist.

You can immediately expand any unconnected port by double-clicking that port in the filtered schematic. When you do so, one level of logic is expanded.

If you would like to expand more than one level of logic, right-click the unconnected port and select 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 select **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). You can also set the **Stop expanding at register** option (which is on by default) to specify whether to stop netlist expansion when a register is reached.

You can select multiple nodes to expand when using the **Expand** command. Note that if you select ports that are located on multiple schematic pages, only the ports on the currently viewed page are shown in the expanded schematic.

The expansion feature works across hierarchical boundaries if the filtered page that contains the unconnected port was generated with the **Filter across hierarchy** option turned on (see “[Filtering in the Schematic View](#)” on page 12–19 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 (see “[Viewing a Timing Path in the Technology Map Viewer](#)” on page 12–26 for details on these schematics).

Reducing a Filtered Netlist

In some cases, you may want to remove logic from a filtered schematic to make the schematic view easier to read or to avoid distraction from logic that you don't need to see on the schematic.

To reduce the filtered schematic, right-click the node or nodes you want to remove and choose the **Reduce** command.

Probing to Source Design File & Other Quartus II Windows

The RTL and Technology Map Viewers provide the ability to cross-probe from the viewer to the source design file and various other windows within the Quartus II software. You can select a node or nodes of interest in the viewer and locate the corresponding node(s) in one of the applicable Quartus II windows, allowing you to view and make changes or assignments in the appropriate editor or floorplan.

To locate, right-click the node or nodes of interest in the schematic and choose the appropriate sub-command from the **Locate** command: **Locate in Assignment Editor**, **Locate in Design File**, **Locate in Timing Closure Floorplan**, **Locate in Chip Editor**, or **Locate in Resource Property Editor**.

The options available for locating depend on the type of node and whether it exists after placement and routing. If the command is enabled in the right button pop-up menu, then it is available for the selected node. The **Locate in Assignment Editor** command is available 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 opens another window in the Quartus II software for the appropriate editor or floorplan and highlights the source of the selected node in that window. You can switch back to the viewer by selecting it in the Window menu or by closing, minimizing, or moving the new window as needed.

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 a node or nodes of interest in another window and locate them in one of the viewers.

You can locate nodes between the RTL and Technology Map Viewers, and you can locate nodes in the RTL Viewer or Technology Map Viewer from the following Quartus II windows:

- Project Navigator
- Timing Closure Floorplan
- Chip Editor
- Resource Property Editor
- Node Finder
- Assignment Editor
- Messages window
- Compilation Report

To locate to 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 name(s) in the **From** or **To** column in the Assignment Editor. Then right-click and choose **Locate > Locate in RTL Viewer** or **Locate in Technology Map Viewer**.

After you choose this command, if the selected node(s) can be found in the viewer, the viewer window is opened or brought into the foreground if already open.



The first time the window is opened after a compilation, the Preprocessor stage runs before the window is opened.

The viewer shows selected node(s) and, if applicable, the connections between those nodes. (The display is similar to that resulting from selecting Filter on Selected Nodes & Nets using Filter Across Hierarchy.) If the node(s) cannot be found in the viewer, a message box is displayed with the message, “Can’t find message location”.

You can switch back to the other Quartus II window by selecting it in the Window menu or by closing, minimizing, or moving the viewer window as needed.

Viewing a Timing Path in the Technology Map Viewer

You can view a timing path from the Timing Analyzer in the Technology Map Viewer. In this case, you can cross-probe from the Compilation Report to see a visual representation of a timing path listed by the Timing Analyzer.

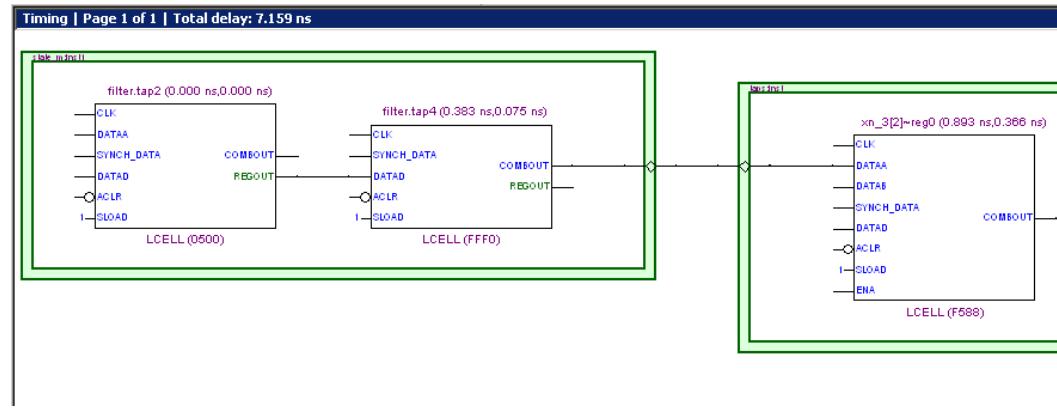
To take advantage of this feature, you must first successfully complete a full compilation of your design, including the Timing Analyzer stage. The **Timing Analyzer** report section of the **Compilation Report** (Processing menu) contains the timing results for your design. You can view a path listed in any of the detailed reports for **Clock Setup: <clock name>**, **tsu**, **tco**, **tpd**, etc. Once you select a detailed report, the timing information is listed in a table format on the right-hand side of the Compilation Report, and each row of this table represents a timing path in the design. To view any particular timing path in the Technology Map Viewer, right-click on the appropriate row in the table and choose **Locate > Locate in Technology Map Viewer**.

The schematic page that is displayed shows the nodes along the timing path with a summary of the total delay as well as timing data representing the interconnect (IC) delay and cell delay 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)**.

Figure 12–16 shows a portion of a timing path represented in the Technology Map Viewer. The total delay for the entire path that goes through a number of levels of logic (only three of which are shown in

Figure 12–16) is 7.159 ns. The delays are indicated for each level of logic, for example the interconnect or 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 diamond symbol on the border indicates the path passes between two different hierarchies.

Figure 12–16. Sample Timing Path Schematic



Other Features in the Schematic Viewer

This section describes other features in the schematic view to enhance usability and help you analyze your design.

Tooltips

Tooltips are displayed whenever the mouse pointer is held over different elements of the schematic. Tooltips contain useful information about nodes, nets, input ports, and output ports.

Table 12–3 lists the information contained in the tooltips for each type of node.

The tooltip information for an instance (the first row in Table 12–3) includes a list of primitives found within that level of hierarchy, and how many of each primitive are contained in the current instance. The number includes all hierarchical blocks below the current instance in the hierarchy. You can get an estimate of the size and complexity of a hierarchical block without navigating into that block.

The tooltip information for atom primitives in the Technology Map Viewer (the third row of [Table 12-3](#)) shows the equation of that design atom. The equations are an expanded version of those that can be viewed in the Equations window in the Timing Closure Floorplan. Advanced users can use these equations to analyze the design implementation in detail.

You can turn off tooltips and change the time that the tooltips are displayed in **Tooltip settings** on the **RTL/Technology Map Viewer** tab of the **Options** dialog box (Tools menu).



For details on understanding equations, refer to Quartus II Help.

Table 12-3. Tooltips Information (Part 1 of 2)

Example Tooltips	Description & Tooltip Format
<pre>taps:inst, INST DFF 32 OPERATOR(SELECTOR) 8 OPERATOR(DECODER) 1</pre>	Instance Format: <instance name>, <instance type> <primitive type>, <number of primitives>... <primitive type>, <number of primitives>
<pre>inst5[3], LCELL (FF00) <r> inst5[3] = DFFEAS((result[?]), GLOBAL(clkx2), VCC,, inst4, , ,)</pre> <pre>acc:inst3 ynm[2]~38, LCELL (0F00) <c> ynm[2]~38 = !filter.tap1 & result[2]</pre>	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.
<pre>clocks:inst7 Mux~1, OPER (MUX)</pre> <pre>md_me:inst18 data[3..3], DFFE</pre>	Primitive Format:<primitive name>, <primitive type>
<pre>pc_clock, INPUT</pre> <pre>Test_probe, OUTPUT</pre>	Pin Format: <pin name>, <pin type>
<pre>node2_OUT1</pre>	Connector Format: <connector name>
<pre>md_me:inst18:dout[15..0], Fanout = 2</pre>	Net Format: <net name>, Fanout = <number of fanout signals>

Table 12–3. Tooltips Information (Part 2 of 2)

Example Tooltips	Description & Tooltip Format
<pre>{Fanout = 23}</pre>	Output port Format: Fanout = <number of fanout signals>
<pre>Source from: reset:reset_irst (1)</pre> <pre>< Destination Index > Source from: < [11] > sample~0:OUT1 < [10] > sample~1:OUT1 < [9] > sample~2:OUT1 < [8] > sample~3:OUT1 < [7] > sample~4:OUT1 < [6] > sample~5:OUT1 < [5] > sample~6:OUT1 < [4] > sample~7:OUT1 < [3] > sample~8:OUT1 < [2] > sample~9:OUT1 < [1] > sample~10:OUT1 < [0] > sample~11:OUT1</pre> (2)	Input port The information displayed depends on the type of the source net. The examples of the tooltips shown represent the following type of source net: (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
<pre>< Destination Index > Source from: < [7..6] > node2:OUT1 < [5] > ct[3]:OUT1 < [4] > node2:OUT1 < [3..2] > ct[3]:OUT1 < [1] > node2:OUT1 < [0] > ct[3]:OUT1</pre> (3)	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).
<pre>< Destination Index > Source from: < [11..0] > 12' h000</pre> (4)	
<pre>< Destination Index > Source from: < [2..1] > 2' h1 < [0] > always7~2:OUT1</pre> (5)	
<pre>< Destination Index > Source from: < [15..0] > md_me:inst18:dout[15:0]</pre> (6)	

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** causes the tooltip to be displayed 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 appears displaying the names of the assigned nodes and pins. Selecting **0** causes the tooltip to appear immediately when the pointer is over an assigned node or pin. Selecting **Unlimited** prevents tooltips from being displayed. The default value is 1 second.

Displaying Net Names

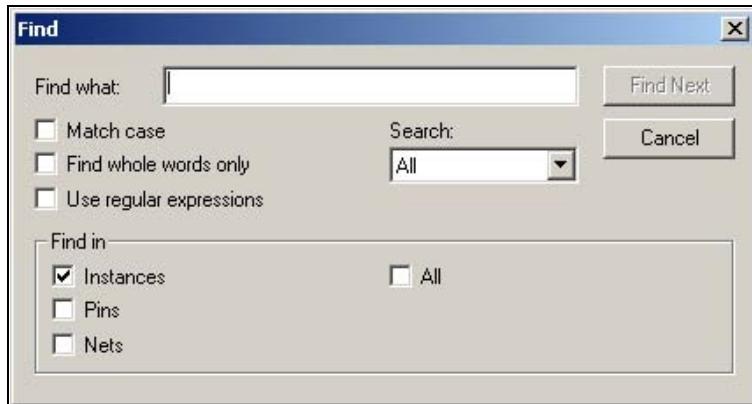
If you would like to see names of all the nets displayed in your schematic, turn on the **Show Net Name** option under **Display Settings** in the **RTL/Technology Map Viewer** page of the **Options** dialog box (Assignments menu). This option is disabled by default. If you turn on the option, the schematic view refreshes automatically to display the net names.

Full Screen View

Set the viewer window to fill the whole screen with the **Full Screen** command (View menu), by clicking the **Full Screen** icon in the viewer toolbar, or by pressing Ctrl+Alt+Space (when you use the keyboard shortcut, return to the standard screen view by pressing Ctrl+Alt+Space again).

Find Command

Open the **Find** dialog box, choose **Find** (View menu) and click **Find** in the viewer toolbar, or right-click in the schematic view and select **Find**. The **Find** dialog box, as shown in [Figure 12-17](#), is the standard search box used throughout the Quartus II software.

Figure 12–17. Find Dialog Box

Select **Up** in the Search list to search from the current hierarchy to upper (parent) hierarchies. Select **Down** to search from the current hierarchy to lower (child) hierarchies.

When you click **Find**, the viewer selects and highlights the first node found, opens the appropriate page of the schematic if necessary, and centers the page so that the node is seen in the viewable area (but does not zoom in to the node). To find the next matching node, click **Find Next**.



See the Quartus II Help for more details on using the **Find** dialog box.

Exporting Schematic as JPEG or BMP Image & Copying to Clipboard

You can export the RTL Viewer or Technology Map Viewer schematic view in JPEG File Interchange Format (**.jpg**) or Windows Bitmap (**.bmp**) file format, allowing you to include the schematic in project documentation or share it with other project members. Export the schematic view with the **Export** command (File menu). In the **Export** dialog box, enter a file name and location and select the desired file type, either JPEG or BMP. The default file name is based on the current instance name, or for pages that involve filtering, expanding, or reducing operations, the default name is “Filter<number of pages exported>”.

You can also copy the schematic to the operating system clipboard with the **Copy** command (Edit menu). You can then paste the schematic into drawing software tools such as Microsoft Paint or Adobe PhotoShop and

save it in another file format such as Graphic Interchange Format (.gif), or paste the schematic into a word processing tool such as Microsoft Word when creating documentation.



In some cases, the schematic view cannot be exported or copied because of limitations in the operating system storage size. In these cases, an error message is generated. To export or copy a schematic that reaches the storage restriction, first split the design into multiple pages (see “[Partitioning the Schematic into Pages](#)” on page 12–15 to control how much of your design is shown on each schematic page).

Printing

To print your schematic page choose **Print** (File menu). You can print each schematic page onto one full page, or you may print the highlighted parts of your schematic onto one page with the **Selected** option. See “[Partitioning the Schematic into Pages](#)” on page 12–15 to control how much of your design is shown on each schematic page.



It may be useful to choose **Page Setup** (File menu) first and change the page orientation from **Portrait** to **Landscape** (or vice versa). You can also adjust the page margins in the **Page Setup** dialog box.

Using the RTL & Technology Map Viewers to Analyze Design Problems

You can use the RTL Viewer and Technology Map Viewer to analyze your design and view how it was interpreted by the Quartus II software. This section provides simple examples of how you can use the RTL and Technology Map Viewers’ capabilities to help analyze real problems you may encounter in the design process.

The RTL Viewer is a good way to view your initial synthesis results to determine whether you have created the desired logic and that it has been implemented correctly in the software. You can use the RTL Viewer to do a visual check of your design before performing a simulation or any other form of verification. Catching design errors at this early stage of the design process can save you valuable time.

If you see unexpected behavior in a part of your design during verification, you can use the RTL Viewer to trace through the initial synthesis netlist and ensure that both the connections and the logic in your design are correct. Viewing the design visually can help you find and analyze the source of problems. If your design looks correct in the RTL Viewer, you know to focus on later stages of the design process for

your analysis, such as optimization during synthesis or place and route, timing problems due to placement and routing, or problems with the verification flow itself.

If you are seeing unexpected synthesis or physical synthesis results, you can use the Technology Map Viewer for a visual representation of the synthesis results (by running the viewer after performing Analysis and Synthesis) or the physical synthesis results (by running the viewer after compiling in the Fitter).

In addition, you can use the RTL Viewer or Technology Map Viewer to locate the source of a particular signal, which can be useful when debugging your design. Use the navigation techniques presented 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 to ensure the connections are as expected.

You can also use the Technology Map Viewer to help you locate post-synthesis nodes of interest in your netlist when making assignments to optimize your design. This functionality can be useful, for example, when making a multicycle clock timing assignment between two registers in your design. It can sometimes be difficult to determine the name of the register that was assigned during synthesis. In the Technology Map Viewer, browse to the desired level of hierarchy using the hierarchy list. Then use the navigation techniques described in this chapter to locate the node. You can start at an I/O port and trace forward or backwards through the design and through levels of hierarchy to find the nodes of interest, or you can locate the register simply by inspecting the schematic.

The RTL Viewer and Technology Map Viewer can be used in many other ways throughout your design, debugging, and optimization stages. Viewing the design netlist is a powerful way of analyzing design problems.

Conclusion

The Quartus II RTL Viewer and Technology Map Viewer allow you to explore and analyze your initial synthesis netlist, post-synthesis netlist, or post-fitter and physical synthesis netlist. The viewers provide a number of features in the hierarchy list and schematic view to help you trace through your netlist and find specific hierarchies or nodes of interest. These capabilities can help you during your design's debugging, optimization, or constraint entry process.

A

Accurate Logic Utilization 9–11
Adaptive Logic Modules
 Architectures with Six-Input LUTs 5–30
Adder 5–28
Adder/Subtractors
 Altera Megafunctions 5–7
AHDL 6–5
ALM 5–30
Altera
 Attribute 6–42
 Specific Attributes 7–11
Altera Megafunctions
 Adder/Subtractors 5–7
altera_chip_pin_lc 7–12
 Other Devices 7–12
 VHDL for ACEX 1K & FLEX 10KE 7–12
altera_implement_in_eab 7–13
altera_implement_in_esb 7–13
altera_io_opendrain 7–13
altera_io_powerup 7–13
Analyze Design Problems
 RTL 12–32
Assignment Editor 1–19
Assignments
 Making 1–18
Asynchronous
 Clear 5–6
 Design 4–3

B

Back-Annotating Node Locations or
 Importing/Exporting 6–15
Binary Multiplexer Design 5–45
Black Box
 Example for Top-Level File A.v 7–37
 in Verilog HDL 7–37
 in VHDL 7–38, 8–27
 Methodology 7–19, 9–16
 Modules
 Including Design Files 8–8

Top-Level File A.vhd 7–38
Top-Level File Verilog 8–26
Top-Level File VHDL 8–27
Verilog HDL 8–26, 9–28
 VHDL 9–28
BLIS
 LogicLock Feature 10–4
 Using Shell Commands 10–9
Block Root 10–3
 Removing 10–9

C

CASE
Assignment
 Default or Others 5–40
Statement
 Degenerate Multiplexer 5–44
 Simple Binary-Encoded 5–39
 Simple One-Hot-Encoded 5–39
Chip Pin 6–44
Clear Box Methodology 7–18, 9–15
Clock
 Clock & Register-Control
 Architectural Features 4–15
 Clock-Gating
 Recommended Method 4–13
 Divided 4–9
 Frequencies 7–7
 Gated 4–11
 Internally-Generated 4–9
 Multiple Clock
 Domains 7–7
 Multiplexed 4–10
 Network Resources 4–15
 Schemes 4–8
Coding Recommendations
 Multiplexers 5–38
Coding Recommendations
 5–31
 Device Specific 5–25

- Combinational
 - Logic Structures **4–4**
 - Loops **4–4**
 - Node/Implement as Output of Logic Cell **6–24**
- Compile
 - Automatically Defining Points from GUI **7–31**
 - Defining Points Using Tcl or SDC **7–30**
 - Manually Defining Points from GUI **7–31**
 - Set Points **7–29**
- Compiler
 - FPGA Compiler II Design Block **10–2**
 - FPGA Compiler II Quartus II Synthesis **10–3**
- Compiler Tool **1–9**
- Compiling **9–5**
- Connectors
 - Input **12–17**
 - Output **12–17**
- Controlling Fan-Out on Data Nets **9–10**
- Copying to Clipboard **12–31**
- Counters
 - Ripple **4–10**
- Create Constraint Files **7–29**
- Cross-Probing **7–15**
 - Enabling **7–15**
 - Quartus II Software **7–15**
- Synplify
 - Software **7–16**
- D**
 - DC FPGA Software
 - Compilation & Synthesis **11–14**
 - Environment for Altera Device Families **11–3**
 - Exporting Designs to the Quartus II Software **11–17**
 - Place & Route with the Quartus II Software **11–19**
 - Quartus II **11–2**
 - Reading Design Files into the Software **11–9**
 - Reporting Design Information **11–15**
 - Saving Synthesis Results **11–16**
 - Selecting a Target Device **11–11**
 - Timing & Synthesis Constraints **11–13**
- dedicated_mult **9–21**
- Defaults
 - Implicit **5–41**
- Degenerate Binary Multiplexer Recoder Design **5–45**
- Delay Chains **4–6**
- Design
 - Compiling **9–6**
 - Creating & Maintaining **10–6**
 - Entry **1–14**
 - Flow **7–1, 8–1, 9–1**
 - Hierarchy **10–1**
 - Optimization **2–22**
 - Recommended Techniques **4–4**
 - Typical Flow **1–2**
- Design Assistant
 - Interface **2–30**
 - Running **2–30**
 - Settings **2–30**
- Design Flow **1–14**
 - RTL **12–3**
- Design Guidelines **4–4**
 - Pulse Generators,Multivibrators **4–6**
- Design Partitioning
 - Hierarchical **4–14**
- Device Support **1–3**
- DSP Block
 - Controlling Inference **8–12, 9–20**
 - Controlling the Inferring **7–23**
 - Guidelines **8–17**
- E**
 - EDA
 - Timing Simulation **1–29**
 - EDIF
 - Creating a Project for Multiple Files **9–30**
 - Creating Multiple Files **8–19, 8–28**
 - Creating Multiple Files Including LogicLock Regions **8–22**
 - Creating Project Including LogicLock Regions **9–25**
 - Generating Design with Multiple Files Using Black Boxes **9–26**
 - Generating Multiple Files **8–24**
 - Generating Multiple Files LogicLock

-
- Option 8–20
 - Manually Creating Multiple Files Using Black Boxes 9–26
 - Editor
 - Assignment 1–19
 - F**
 - Filtered Netlist 12–21
 - Across Hierarchies 12–22
 - Expanding 12–24
 - Reducing 12–24
 - FSM
 - Explorer in Synplify Pro 7–10
 - Finite State Machine (FSM) Compiler 7–9
 - full_case Attribute 6–22
 - G**
 - General Optimization
 - Attributes 7–10
 - Options 7–10
 - H**
 - HardCopy
 - Compile, FPGA Design 2–16
 - Design Flow 2–14
 - One Step Process 2–16
 - Design Guidelines, Checking Designs 2–29
 - Devices
 - Designing 2–16
 - Floorplans 2–22
 - FPGA Project
 - Close 2–16
 - Generating Design Database 2–31
 - Open Project 2–16
 - Power Calculator
 - APEX 20K 2–33
 - Stratix 2–34
 - Power Estimator
 - APEX 2–35
 - Stratix 2–33
 - Project, Migrate Compiled 2–16
 - Stratix Device
 - Compile 2–16
 - Stratix Devices 2–12
 - Targeting HardCopy APEX Devices 2–36
 - Timing Optimization Wizard 2–19
 - HardCopy II
 - Design Benefits 2–1
 - Device Resource Guide 2–4
 - Planning 2–2
 - Prototyping Flow 2–2
 - Hazards 4–3
 - HDL
 - Altera Recommended Coding Guidelines 2–29
 - Inferring Adder/Subtractors 5–7
 - Inferring Altera Megafunctions 7–22, 11–7
 - Inferring Counters 5–5
 - Inferring RAM 5–14
 - Inferring ROM 5–19
 - Inferring Shift Registers 5–22
 - Inferring Multipliers 5–8
 - Inferring Multiply-Accumulators Multiply-Adders 5–11
 - Options in Source Code 6–40
 - Read Comments 6–21
 - Hierarchy
 - Considerations 10–5
 - Design Considerations 8–18, 9–23
 - List
 - 12–13
 - Selecting an Item 12–14
 - Preserving 7–11
 - I**
 - I/O
 - Assigning Registers 9–9
 - Disabling I/O
 - Pad Insertion 9–9
 - Flip-Flops 6–40
 - Input/Output Delays 7–8
 - Preventing Precision RTL Synthesis from Adding I/O Pad On Individual Pin 9–10
 - Preventing Precision RTL Synthesis from Adding I/O Pads 9–10
 - Register
 - Mapping 8–6
 - Setting
 - Assigning

- Pin Numbers [9–8](#)
- IF Statement
 - Default Conditions Explicitly Specified [5–42](#)
 - Implicit Defaults [5–42](#)
 - Implying Priority [5–40](#)
- Incremental Synthesis
 - Block-Level [10–2](#)
 - Flow [8–29](#)
 - Preparing a Design [6–53](#)
 - Synthesizing a Design [6–54](#)
 - Synthesizing Using the Automatic Compilation Flow [6–54](#)
 - Synthesizing Using the Synthesis & Merge Commands [6–54](#)
- Inferred Counters, Adder-Subtractors, Shift Registers, Memory, & DSP Functions [6–49](#)
- Inferring
 - Quartus II Megafunction from HDL Code [9–17](#)
 - Quartus II Memory Elements [8–10](#)
 - RAM [8–10](#)
 - ROM [8–11](#)
- Instantiating
 - Altera Megafunction
 - Using the MegaWizard Plug-In Manager [11–5](#)
- L**
 - Labeling Block Roots [10–7](#)
 - Latches [4–5, 5–31](#)
 - Logic
 - Cells
 - Remove Redundant [6–34](#)
 - Elements
 - Architectures with Four-Input LUTs [5–29](#)
 - Remove Duplicate [6–33](#)
 - LogicLock
 - Apply Attributes [7–32](#)
 - Block-Based Design [9–23](#)
 - Creating Design with Separate Blocks [9–24](#)
 - Methodology [6–15](#)
 - Block-Based Design [7–27](#)
 - Block-based Design [8–18](#)
 - LogicLock_Incremental.tcl
 - Modifications Required for Script File [8–30](#)
- LogicLock_Interface.tcl
 - Incremental Synthesis for Script File [8–30](#)
- Look & Feel
 - MAX+PLUS II [1–7](#)
- M**
 - MAX+PLUS II
 - Converting
 - Existing Design [1–11](#)
 - Graphic Design Files [1–12](#)
 - Design Conversion [1–11](#)
 - Importing Assignments [1–13](#)
 - Look & Feel [1–6](#)
 - Maximum Fan Out [7–10](#)
 - Megafunction
 - Architecture-Specific Features [9–14, 11–4](#)
 - Inference Control [6–35](#)
 - Megafunction Wizard
 - Variation Wrapper Files [11–5](#)
 - Variation Wrapper Files Black Box
 - Methodology [11–6](#)
 - Verilog HDL Files Black Box
 - Instantiation [11–6](#)
 - Module
 - Level Attributes [8–13](#)
 - Modules Constraint Table
 - Opening Block Roots [10–7](#)
 - Multiplexers
 - Binary [5–38](#)
 - Buses [5–46](#)
 - Coding Recommendations [5–38](#)
 - Degenerate [5–43](#)
 - Priority [5–38, 5–39](#)
 - Restructure [6–30](#)
 - Restructuring Option [5–47](#)
 - Selector [5–39](#)
 - Multiplier
 - [7–22, 9–17](#)
 - Accumulators [8–12, 9–19](#)
 - Adders [8–12, 9–19](#)
 - Inferring DSP Functions [8–11](#)
 - Simple [8–12](#)
 - Multiplier Style [6–38](#)
 - Multiply Accumulators
 - Multiply-Adders [6–35](#)
 - Multiply-Adders [5–11](#)

N	Q
NativeLink Integration	Quartus II
Exporting Designs 8–8, 9–12	Attributes 6–42, 7–11
Exporting Quartus II Designs 7–13	Compile, Time 3–2
Net Names	Constraints
Displaying 12–30	LAB Assignments 2–27
Netlist Files	Location 2–27
Exporting Block-Level 10–7	Placement 2–26
Generating 8–8	Design Flow 6–6
Nets	ECO Support 3–2
Preserving 7–10	HDL Level 3–3
Node Finder 1–4	Netlist Level 3–5
Node-Naming Conventions	Features for HardCopy II Planning 2–2
Combinational Logic Cells 6–49	Functional Simulation 1–21
Registers (DFF or D Flip-Flop Atoms) 6–47	Hierarchy
O	Hierarchical Boundary
Obtaining Accurate Logic Utilization & Timing	Preserve 6–30
Analysis Reports 9–11	Integration 8–9
Optimization Strategies 8–4	LogicLock
Other Features 12–25	Assignments 2–28
P	Migrated HardCopy Stratix .qsf File 2–29
parallel_case Attribute 6–24	Supported Constraints Example 2–28
Passing Constraints Via Scripts 8–8	Maximum Fan-Out 6–26
Paths	Megafunction
False 7–8	Architecture-Specific Features 7–17
Multi-Cycle Paths 7–8	Inferring from HDL
Performance Estimation 2–22	7–22
Power Tab	Inferring from HDL Code
Clock	5–4
8–4	Instantiating 8–9
Global 8–4	Instantiating and Inferring 5–1
Input & Output 8–5	Instantiating in HDL Code 5–2
Power-Up	Instantiating Using MegaWizard Plug-In
Don't Care 6–33	Manager 5–2
Level 6–32	9–15
preserve_signal 9–21	Instantiating Using MegaWizard Plug-In
Project	Manager
Creating 9–5	7–18
Creating New 1–14	Instantiating Using Port &
Project Navigator 1–4	Parameter 5–4
Pulse Generators,Multivibrators	LPM Functions 8–9
Design Guidelines 4–6	MegaWizard
	Using Generated Verilog HDL Files for
	Black-Box Megafunction
	Instantiation 7–19
	Using Generated Verilog HDL Files for
	Clear Box Megafunction
	Instantiation 7–18

- Using Generated VHDL Files for Black Box Megafunction
 - Instantiation 7–20
 - Optimization Technique 6–28
 - Place & Route 1–23
 - Power Estimation 1–29, 2–33
 - Preserve Registers 6–25
 - Probing to Source Design File 12–25
 - Quick Menu Reference 1–31
 - RAM and ROM 6–36
 - Running the Software Manually 9–14
 - State Machine Processing 6–28
 - Synthesis 1–21
 - Attributes 6–17
 - Considerations & Restrictions 6–13
 - Directives 6–17
 - Flow
 - Incremental 8–29
 - Forcing Complete Re-synthesis 6–12
 - Formal Verification Tools 6–15
 - Incremental 6–8
 - Options 6–16, 6–51
 - Partitions 6–8
 - Preparing a Design 6–10
 - Preserve Hierarchical Boundary Logic Option 6–15
 - Resource Balancing 6–14
 - Restrictions on Megafunction Partitions 6–14
 - Synthesizing
 - Design 6–10
 - Using the Automatic Compilation Flow 6–10
 - Using the Synthesis & Merge Commands 6–11
 - Virtual I/O Pins 6–16
 - Hierarchical Considerations 6–13
 - Tcl
 - HardCopy Stratix Support 2–35
 - Translate Off & On 6–20
 - Quartus II
 - Command Reference for MAX+PLUS II 1–33
 - GUI Overview 1–4
 - Integrated Synthesis
 - Node-Naming Conventions 6–46
 - Logic Options 6–19
 - Setting Other Options in Your HDL Source Code 6–40
 - Setting up MAX+PLUS II 1–6
 - Simulator Tool 1–27
 - Synthesis Options 6–20
- R**
- RAM 7–25
 - Style 6–37
 - RAM & DSP Block
 - Input Output Registers 6–49
 - Register
 - Control Signals 4–17
 - Packing 7–11
 - Remove Duplicate 6–34
 - Secondary Control Signals Flip-Flops 5–25
 - Reports, Summary 2–30
 - Reset Resources 4–17
 - Resource Balancing 7–23
 - ROM
 - Inferring 7–27
 - RTL Viewer 12–1
 - Overview 12–1
- S**
- Sample Verilog-1995
 - Code with a ramstyle Attribute 6–37
 - Schematic
 - Design Entry 6–6
 - Exporting as JPEG or BMP 12–31
 - Filtering 12–19
 - Following Nets Between Pages 12–17
 - Moving Between Pages 12–16
 - Navigating View 12–14
 - Other Features in Viewer 12–27
 - Partitioning into Pages 12–15
 - Selecting Item in View 12–12
 - Symbols 12–5
 - View 12–5
 - Setting
 - Constraints 9–6
 - Mapping Constraints 9–7
 - Timing Constraints 9–7
 - Shift Registers 5–22
 - Signal
 - Attributes for Controlling DSP Block Infer-

-
- ence for VHDL Code 8–16
 - Attributes for Controlling DSP Block Inference in Verilog HDL Code 8–16
 - Level Attributes 8–15
 - Signal Level Attribute 7–23
 - Signals
 - Tri-State
 - Verilog HDL 5–28
 - VHDL 5–28
 - Signals
 - Tri-State 5–28
 - SignalTap II
 - Logic Analyzer 6–16
 - Single Precision Project 9–24
 - Source
 - Changing Within a Block 10–8
 - State Machines 5–32
 - Stratix Devices
 - HardCopy 2–12
 - Summary Reports 2–30
 - Synchronous
 - Clock Enables 4–12
 - Design Fundamentals 4–2
 - FPGA Design Practices 4–1
 - Synplify
 - Optimization Strategies 7–5
 - Pro
 - Implementations in 7–6
 - Software
 - Attributes for Black-Boxing 7–21
 - Launch 7–14
 - Running Quartus II from Within 7–14
 - Synplify Pro
 - Retiming 7–11
 - Synthesis
 - Attributes,Quartus II 6–17
 - Directives,Quartus II 6–17
 - Options,Quartus II 6–16
 - Registers That Can Change 6–48
 - Synthesizing the Design & Evaluating the Results 9–11
- ## T
- Tcl
 - Console 1–4
 - HardCopy Migration Support 2–21
 - Running Script File in
- LeonardoSpectrum 8–31
 - Specifying a Destination Library Name in the QSF 6–4
 - Technology Map Viewer 12–2
 - Overview 12–2
 - Technology Map Viewers 12–3, 12–32
 - Time Stamp Synthesis 10–6
 - Timing
 - Analysis 1–24
 - Leonardo-Spectrum Software 8–7
 - Analysis Reports 9–11
 - Assignments 1–20
 - Closure Floorplan 1–26
 - Driven Synthesis 8–4
 - Models 2–22
 - Simulation 1–27
 - Static Analysis 2–33
 - Synthesis Settings 7–6
 - Viewing Path in Technology Map Viewer 12–26
 - Traversing Design Hierarchy 12–18
 - Trees
 - Trees 5–28
 - Typical Flow
 - Design 1–2
- ## U
- Using Black Box 8–24
- ## V
- Verilog HDL
 - 6–1, 9–18, 9–21
 - 64-Bit Long Shift Register 5–23
 - 64-Bit Long Shift Register 5–23
 - Adder/Subtractor 5–7
 - Code with a 6–22, 6–24
 - Code with a useioff Attribute 6–41
 - Counter with Count Enable 5–6
 - dedicated_mult 9–21
 - D-Flip-Flop with Control Signals 5–27
 - Dual-Clock Synchronous RAM 5–16
 - Example 7–21
 - Module Level Attributes 8–14
 - Multiplier Implemented in Logic 9–18
 - Pipelined Binary Tree 5–29
 - Pipelined Ternary Tree 5–30

- Read Comments as HDL Example 6–21
- Signal Attributes for Controlling DSP Block
 - Inference 7–24
- Signed Multiplier 5–10
- Signed Multiply-Adder 5–12
- Simple Binary-Encoded “Case” Statement 5–39
- Simple One-Hot-Encoded “Case” Statement 5–39
- Single-Clock Synchronous RAM 5–15
- State Machine Coding Example 5–34
- State Machines 5–33
- Synchronous ROM 5–21
- Top-level Code with Black Box Instantiation 7–20
- Translate Off & On Example 6–20
- Unsigned Multiplier 5–9
- Unsigned Multiply Accumulator 5–11
- Using MegaWizard-generated Files for Black Box Megafunction Instantiation 9–16
- Using MegaWizard-generated Files for Clear Box Megafunction Instantiation 9–15
- Verilog HDL Example of an Unsigned Multiplier 5–9
- Verilog HDL
 - Support 6–1
- Verilog-1995
 - Chip Pin to a Bus of Pins 6–45
 - Chip Pin to a Single Pin 6–45
 - HDL 6–18
 - Quartus II Attribute an Entity 6–44
 - Quartus II Attribute an Instance 6–43
- Verilog-2001 6–25
 - Chip Pin to a Single Pin 6–45
 - Chip Pin to Part of a Bus of Pins 6–45
 - HDL 6–19
 - Quartus II Attribute an Entity 6–44
 - Quartus II Attribute an Instance 6–43
- VHDL 6–2
 - 9–18, 9–21
 - 64-Bit Long Shift Register 5–24
 - Adder/Subtractor 5–8
 - Chip Pin to Part of a Bus of Pins 6–45
 - Code for syn_encoding 7–9
 - Code with a ramstyle Attribute 6–37
 - Code with a useioff Attribute 6–41
 - Counter with Synchronous Load 5–6, 5–16
- D-Flip-Flop with Control Signals 5–27
- Dual-Clock Synchronous RAM 5–17
- extract_mac 9–21
- Inferred Dual-Port RAM 7–25
- Inferred Dual-Port RAM Preventing Bypass Logic 7–26
- Libraries 6–3
- Module Level Attributes 8–14
- Multiplier Implemented in Logic 9–19
- Preventing Unintentional Latch Creation 5–32
- Quartus II Attribute an Entity 6–44
- Quartus II Attribute an Instance 6–43
- Read Comments as HDL Example 6–21
- Signal Attributes for Controlling DSP Block
 - Inference 7–24
- Signed Multiplier 5–9
- Signed Multiply Accumulator 5–14
- Single-Clock Synchronous RAM with Asynchronous Read Address 5–19
- State Machine Example 5–36
- State Machines 5–37
- Synchronous ROM 5–21
- Top-level Code with Black Box Instantiation 7–21
- Translate Off & On Example 6–20
- Unsigned Multiplier 5–10
- Unsigned Multiply-Adder 5–13
- Using MegaWizard-generated Files for Black Box Megafunction Instantiation 9–16
- Using MegaWizard-generated Files for Clear Box Megafunction Instantiation 9–16
- Specifying a Destination Library Name 6–4
- VQM
 - Creating a Design with Multiple Files 7–28
 - Creating a Design with Multiple Files using Multipoint Synthesis 7–29
 - Creating a Quartus II Project for Multiple Files 7–34, 7–39
 - Generating a Design with Multiple Files Using Black Boxes 7–35
 - Hierarchy & Design Considerations with Multiple Files 7–28
 - Manually Creating Multiple Files Using Black Boxes 7–35



Quartus II Handbook, Volume 2

Design Implementation & Optimization

ALTERA[®]

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
<http://www.altera.com>

Copyright © 2005 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.



Printed on recycled paper



Chapter Revision Dates	xi
-------------------------------------	-----------

About this Handbook	xiii
----------------------------------	-------------

How to Contact Altera	xiii
Typographic Conventions	xiii

Section I. Scripting & Constraint Entry

Revision History	Section I-2
------------------------	-------------

Chapter 1. Assignment Editor

Introduction	1-1
Using the Assignment Editor	1-1
Effects of Settings Made Outside the Assignment Editor User Interface	1-2
Category, Node Filter, Information, Edit Bars & Spreadsheet	1-3
Category Bar	1-3
Node Filter Bar	1-5
Information Bar	1-6
Edit Bar	1-7
Assignment Editor Features	1-8
Using the Enhanced Spreadsheet Interface	1-8
Dynamic Syntax Checking	1-9
Node Filter Bar	1-10
Using Time Groups	1-11
Customizable Columns	1-12
Tcl Interface	1-13
Creating Pin Locations Using the Assignment Editor	1-13
Setting Pin Locations From the Pin Number List	1-13
Setting Pin Locations From the Design Pin Name List	1-14
Exporting & Importing Assignments	1-16
Exporting Assignments	1-16
Importing Assignments	1-18
Conclusion	1-20

Chapter 2. Command-Line Scripting

Introduction	2-1
The Benefits of Command-Line Executables	2-1
Introductory Example	2-2
Design Flow	2-3

Text-Based Report Files	2-6
Compilation with quartus_sh --flow	2-7
Command-Line Scripting Help	2-7
Command-Line Option Details	2-9
Option Precedence	2-9
Command-Line Scripting Examples	2-11
Create a Project & Apply Constraints	2-11
Check Design File Syntax	2-13
Create a Project & Synthesize a Netlist Using Netlist Optimizations	2-14
Attempt to Fit a Design as Quickly as Possible	2-14
Fit a Design Using Multiple Seeds	2-14
Makefile Implementation	2-16
The QFlow Script	2-18
Conclusion	2-20

Chapter 3. Tcl Scripting

Introduction	3-1
What is Tcl?	3-1
Tcl Scripting Basics	3-2
Hello World Example	3-2
Variables	3-3
Substitutions	3-3
Arithmetic	3-4
Lists	3-4
Arrays	3-5
Control Structures	3-5
Procedures	3-6
File I/O	3-7
Syntax & Comments	3-8
Quartus II Tcl API Help	3-8
Quartus II Tcl Packages	3-9
Loading Packages	3-10
Executables Supporting Tcl	3-11
Command-Line Options: -t, -s, and --tcl_eval	3-11
Run a Tcl Script	3-12
Interactive Shell Mode	3-12
Evaluate as Tcl	3-12
Using the Quartus II Tcl Console Window	3-12
Examples	3-13
Natural Bus Naming	3-13
Accessing Command-Line Arguments	3-13
Using the cmdline Package	3-14
Creating Projects & Making Assignments	3-16
Compiling Designs	3-17
Extracting Report Data	3-18
Using Collection Commands	3-19

Timing Analysis	3–20
EDA Tool Assignments	3–22
Using LogicLock Regions	3–24
Using the post_message Command	3–27
Using the Quartus II Tcl Shell in Interactive Mode	3–28
Getting Help on Tcl & Quartus II Tcl APIs	3–31
Quartus II Legacy Tcl Support	3–33
References	3–34

Chapter 4. Quartus II Project Management

Introduction	4–1
Using Revisions With Your Design	4–1
Creating & Deleting Revisions	4–2
Comparing Revisions	4–5
Creating Different Versions of Your Design	4–5
Archiving Projects With the Quartus II Archive Project Feature	4–6
Version-Compatible Databases	4–9
Quartus II Project Platform Migration	4–10
Filenames and Hierarchy	4–10
Specifying Libraries	4–12
Search Path Precedence Rules	4–13
Quartus II-Generated Files for 3rd Party EDA Tools	4–13
Migrating Database Files	4–13
Scripting Support	4–13
Managing Revisions	4–14
Archiving Projects With a Tcl Command or at the Command Prompt	4–14
Restoring Archived Projects	4–15
Importing & Exporting Version-Compatible Databases	4–15
Specifying Libraries	4–16
Conclusion	4–16

Section II. Device & Board Utilities

Revision History	Section II–1
------------------------	--------------

Chapter 5. I/O Assignment Planning & Analysis

Introduction	5–1
I/O Assignment Planning & Analysis Design Flows	5–1
Design Flow Without Design Files	5–2
Design Flow With Design Files	5–4
I/O Rules Checked by the I/O Assignment Analysis	5–6
Inputs for I/O Assignment Analysis	5–8
Reserving Pins	5–8
Location Assignments	5–9

Suggested & Partial Placement	5–10
Generating a Mapped Netlist	5–10
Understanding the I/O Assignment Analysis Report & Messages	5–11
Scripting Support	5–12
Running the I/O Assignment Analysis	5–12
Reserving Pins	5–13
Location Assignments	5–13
Generating a Mapped Netlist	5–13
Conclusion	5–14

Section III. Area Optimization & Timing Closure

Revision History	Section III–2
------------------------	---------------

Chapter 6. Design Optimization for Altera Devices

Introduction	6–1
Stages in the Compilation Process Described in this Chapter	6–1
Design Space Explorer	6–2
Optimization Advisors	6–2
Initial Compilation	6–3
Device Setting	6–3
Timing Requirements Settings	6–4
Smart Compilation Setting	6–5
Early Timing Estimation	6–5
Timing-Driven Compilation Settings	6–5
Fitter Effort Setting	6–7
I/O Assignments	6–7
Design Analysis	6–8
Resource Utilization	6–9
I/O Timing (Including t_{PD})	6–10
f_{MAX} Timing	6–11
Compilation Time	6–13
Resource Utilization Optimization Techniques (LUT-Based Devices)	6–14
Resolving Resource Utilization Issues Summary	6–14
Use Register Packing	6–16
Remove Fitter Constraints	6–19
Perform WYSIWYG Resynthesis for Area	6–19
Optimize Synthesis for Area, Not Speed	6–20
Change State Machine Encoding	6–20
Flatten the Hierarchy During Synthesis	6–21
Restructure Multiplexers	6–21
Retarget Memory Blocks	6–22
Retarget or Balance DSP Blocks	6–22
Optimize Source Code	6–23
Modify Pin Assignments or Choose a Larger Package	6–24
Use a Larger Device	6–24
I/O Timing Optimization Techniques (LUT-Based Devices)	6–24
Improving Setup & Clock-to-Output Times Summary	6–25

Timing-Driven Compilation	6-25
Fast Input, Output & Output Enable Registers	6-26
Programmable Delays	6-27
Use PLLs to Shift Clock Edges	6-29
Use Fast Regional Clocks in Stratix Devices	6-30
Change How Hold Times Are Optimized for MAX II Devices	6-30
f _{MAX} Timing Optimization Techniques (LUT-Based Devices)	6-31
Improving f _{MAX} Summary	6-31
Synthesis Netlist Optimizations & Physical Synthesis Optimizations	6-32
Seed	6-33
Optimize Synthesis for Speed, Not Area	6-34
Flatten the Hierarchy During Synthesis	6-35
Set the Synthesis Effort to High (Where Applicable)	6-35
Change State Machine Encoding	6-36
Duplicate Logic for Fan-Out Control	6-36
Use Other Synthesis Options Available in Your Synthesis Tool	6-37
LogicLock Assignments	6-37
Location Assignments & Back-Annotation	6-40
Optimize Source Code	6-45
Resource Utilization Optimization Techniques (Macrocell-Based CPLDs)	6-46
Use Dedicated Inputs for Global Control Signals	6-46
Reserve Device Resources	6-47
Pin Assignment Guidelines & Procedures	6-47
Resolving Resource Utilization Problems	6-50
Timing Optimization Techniques (Macrocell-Based CPLDs)	6-54
Improving Setup Time	6-55
Improving Clock-to-Output Time	6-56
Improving Propagation Delay (tPD)	6-57
Improving Maximum Frequency (f _{MAX})	6-58
Optimizing Source Code—Pipelining for Complex Register Logic	6-59
Compilation-Time Optimization Techniques	6-60
Reducing Synthesis & Synthesis Netlist Optimization Time	6-60
Check Early Timing Estimation Before Fitting	6-61
Reducing Placement Time	6-62
Reducing Routing Time	6-64
Scripting Support	6-64
Initial Compilation Settings	6-65
Resource Utilization Optimization Techniques (LUT-Based Devices)	6-65
I/O Timing Optimization Techniques (LUT-Based Devices)	6-66
f _{MAX} Timing Optimization Techniques (LUT-Based Devices)	6-67
Conclusion	6-68

Chapter 7. Timing Closure Floorplan

Introduction	7-1
Design Analysis Using the Timing Closure Floorplan	7-1
Timing Closure Floorplan Views	7-2
Viewing Assignments	7-3

Viewing Critical Paths	7–6
Physical Timing Estimates	7–10
LogicLock Region Connectivity	7–12
Viewing Routing Congestion	7–14
I/O Timing Analysis Report File	7–15
f_{MAX} Timing Analysis Report File	7–18
Conclusion	7–21

Chapter 8. Netlist Optimizations & Physical Synthesis

Introduction	8–1
Synthesis Netlist Optimizations	8–2
WYSIWYG Primitive Resynthesis	8–2
Gate-Level Register Retiming	8–4
Preserving Synthesis Netlist Optimization Results	8–8
Physical Synthesis Optimizations	8–9
Physical Synthesis for Combinational Logic	8–10
Physical Synthesis for Registers—Register Duplication	8–11
Physical Synthesis for Registers—Register Retiming	8–13
Preserving Your Physical Synthesis Results	8–13
Applying Netlist Optimization Options	8–15
Scripting Support	8–15
Synthesis Netlist Optimizations	8–16
Physical Synthesis Optimizations	8–17
Back-Annotating Assignments	8–17
Conclusion	8–18

Chapter 9. Design Space Explorer

Introduction	9–1
DSE Concepts	9–1
DSE Exploration	9–2
General Description	9–2
DSE Flow	9–4
DSE Support for Altera Device Families	9–5
DSE Project Settings	9–6
Setting Up the DSE Working Environment	9–6
Specifying the Revision	9–6
Setting the Initial Seed	9–6
Restructuring LogicLock Regions	9–6
Effort Search for Quartus Integrated Synthesis Projects	9–7
Performing an Advanced Search in Design Space Explorer	9–8
Exploration Space	9–8
Optimization Goal	9–12
Search Method	9–13
DSE Flow Options	9–13
Create a Revision From a DSE Point	9–13
Continue Exploration Even if Base Compilation Fails	9–15
Run Quartus Assembler During Exploration	9–15

Archive All Compilations	9–15
Save Exploration Space to File	9–15
Stop Flow After Time	9–15
DSE Advanced Information	9–16
Computer Load Sharing in DSE Using Distributed Exploration Searches	9–16
Creating Custom Spaces for DSE	9–17
Conclusion	9–21

Chapter 10. LogicLock Design Methodology

Introduction	10–1
Improving Design Performance	10–1
Block-Based Design with the Quartus II LogicLock Methodology	10–2
Preserving Timing Results Using the LogicLock Flow	10–3
Creating LogicLock Regions	10–4
Timing Closure Floorplan View	10–10
LogicLock Region Properties	10–11
Hierarchical (Parent and/or Child) LogicLock Regions	10–12
Assigning LogicLock Region Content	10–13
Tcl Scripts	10–16
Quartus II Block-Based Design Flow	10–17
Additional Quartus II LogicLock Design Features	10–23
LogicLock Restrictions	10–31
Constraint Priority	10–31
Placing LogicLock Regions	10–31
Placing Memory, Pins & Other Device Features into LogicLock Regions	10–32
Back-Annotating Routing Information	10–34
Exporting Back-Annotated Routing in LogicLock Regions	10–34
Importing Back-Annotated Routing in LogicLock Regions	10–36
Scripting Support	10–37
Initializing & Uninitializing a LogicLock Region	10–38
Creating or Modifying LogicLock Regions	10–38
Obtaining LogicLock Region Properties	10–38
Assigning LogicLock Region Content	10–39
Prevent Further Netlist Optimization	10–39
Save a Node-level Netlist into a Persistent Source File (.vqm)	10–39
Exporting LogicLock Regions	10–40
Importing LogicLock Regions	10–40
Setting LogicLock Assignment Priority	10–40
Assigning Virtual Pins	10–41
Back-Annotating LogicLock Regions	10–41
Conclusion	10–41

Chapter 11. Synplicity Amplify Physical Synthesis Support

Software Requirements	11–1
Amplify Physical Synthesis Concepts	11–1
Amplify-to-Quartus II Flow	11–2
Initial Pass: No Physical Constraints	11–3

Iterative Passes: Optimizing the Critical Paths	11-5
Using the Amplify Physical Optimizer Floorplans	11-5
Multiplexers	11-7
Independent Paths	11-8
Feedback Paths	11-9
Starting and Ending Points	11-9
Utilization	11-11
Detailed Floorplans	11-11
Forward Annotating Amplify Physical Optimizer Constraints into the Quartus II Software	
11-12	
Altera Megafunctions Using the MegaWizard Plug-In Manager with the Amplify Software	
11-13	
Conclusion	11-14

Index



Chapter Revision Dates

The chapters in this book, *Quartus II Handbook, Volume 2*, were revised on the following dates. Where chapters or groups of chapters are available separately, part numbers are listed.

Chapter 1. Assignment Editor

Revised: December 2004
Part number: *qii52001-2.1*

Chapter 2. Command-Line Scripting

Revised: December 2004
Part number: *qii52002-2.1*

Chapter 3. Tcl Scripting

Revised: December 2004
Part number: *qii52003-2.2*

Chapter 4. Quartus II Project Management

Revised: February 2004
Part number: *qii52012-1.1*

Chapter 5. I/O Assignment Planning & Analysis

Revised: January 2005
Part number: *qii52004-2.1*

Chapter 6. Design Optimization for Altera Devices

Revised: December 2004
Part number: *qii52005-2.1*

Chapter 7. Timing Closure Floorplan

Revised: December 2004
Part number: *qii52006-2.1*

Chapter 8. Netlist Optimizations & Physical Synthesis

Revised: December 2004
Part number: *qii52007-2.1*

Chapter 9. Design Space Explorer

Revised: December 2004
Part number: *qii52008-2.1*

Chapter 10. LogicLock Design Methodology

Revised: *December 2004*Part number: *qii52009-2.2*

Chapter 11. Synplicity Amplify Physical Synthesis Support

Revised: *June 2004*Part number: *qii52011-1.0*



About this Handbook

This handbook provides comprehensive information about the Altera® Quartus®II design software, version 4.2.

How to Contact Altera

For the most up-to-date information about Altera products, go to the Altera world-wide web site at www.altera.com. For technical support on this product, go to www.altera.com/mysupport. For additional information about Altera products, consult the sources shown below.

Information Type	USA & Canada	All Other Locations
Technical support	www.altera.com/mysupport/	altera.com/mysupport/
	(800) 800-EPLD (3753) (7:00 a.m. to 5:00 p.m. Pacific Time)	(408) 544-7000 (1) (7:00 a.m. to 5:00 p.m. Pacific Time)
Product literature	www.altera.com	www.altera.com
Altera literature services	(1)	(1)
Non-technical customer service	(800) 767-3753	(408) 544-7000 (7:30 a.m. to 5:30 p.m. Pacific Time)
FTP site	ftp.altera.com	ftp.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 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: f_{MAX} , \qdesigns directory, d: drive, chiptrip.gdf file.
<i>Italic Type with Initial Capital Letters</i>	Document titles are shown in italic type with initial capital letters. Example: <i>AN 75: High-Speed Board Design</i> .

Visual Cue	Meaning
<i>Italic type</i>	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	The caution indicates required information that needs special consideration and understanding and should be read prior to starting or continuing with the procedure or process.
	The warning indicates information that should be read prior to starting or continuing the procedure or processes
↔	The angled arrow indicates you should press the Enter key.
→→	The feet direct you to more information on a particular topic.



Section I. Scripting & 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 Floorplan Editor to specify your initial design constraints, such as pin assignments, device options, logic options, and timing constraints.

This section describes how to take advantage of these components of 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.

This section includes the following chapters:

- Chapter 1, Assignment Editor
- Chapter 2, Command-Line Scripting
- Chapter 3, Tcl Scripting
- Chapter 4, Quartus II Project Management

Revision History

The table below shows the revision history for [Chapter 1](#), [2](#), [3](#), and [4](#).

Chapter(s)	Date / Version	Changes Made
1	Dec. 2004 v2.1	Updated for Quartus II software version 4.2: <ul style="list-style-type: none"> ● General formatting and editing updates. ● Updated information about refreshing the Assignment Editor on page 1–2. ● Updated Figures 1–1, 1–2, 1–3, 1–5, 1–9, and 1–11. ● Added information about how to make selections to the Assignment Editor window on page 1–6. ● Added a reference for more information on Time Groups on page 1–11. ● Reworded description of “Customizable Columns”. ● Added new section “Creating Pin Locations Using the Assignment Editor”. ● Added new description to “Exporting & Importing Assignments”.
	June 2004 v2.0	<ul style="list-style-type: none"> ● Updates to tables, figures. ● New functionality in the Quartus II software version 4.1.
	Feb. 2004 v1.0	Initial release.
2	Dec. 2004 v2.1	<ul style="list-style-type: none"> ● Updates to tables and figures. ● New functionality in the Quartus II software version 4.2.
	June 2004 v2.0	<ul style="list-style-type: none"> ● Updates to tables and figures. ● New functionality in the Quartus II software version 4.1.
	Feb. 2004 v1.0	Initial release.
3	Dec. 2004 v2.1	<ul style="list-style-type: none"> ● Updates to tables and figures. ● New functionality in the Quartus II software version 4.2.
	Aug. 2004 v2.1	<ul style="list-style-type: none"> ● Minor typographical corrections ● Enhancements to example scripts.
	June 2004 v2.0	<ul style="list-style-type: none"> ● Updates to tables, figures. ● New functionality in the Quartus II software version 4.1.
	Feb. 2004 v1.0	Initial release.

Chapter(s)	Date / Version	Changes Made
4	Dec. 2004 v1.1	<p>Updated for Quartus II software version 4.2:</p> <ul style="list-style-type: none">● General formatting and editing updates.● Added new Figures 4–1 and 4–2.● 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 4–3 and 4–5.● Added new information about displaying assignments for multiple revisions on page 4–5.● Updated Steps 5 and 6 in “Archive a Project”.● Updated Step 6 in “Restore an Archived Project”.● “Version-Compatible Databases” describes migration to Quartus II software version 4.2.● Corrected Tcl commands in “Importing & Exporting Version-Compatible Databases”.
	June 2004 v1.0	Initial release.

qii52001-2.1

Introduction

As a result of the increasing complexity of today's FPGA designs and the demand for higher performance, designers must make a larger number of complex timing and logic constraints to meet performance requirements. This complexity 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
- Effects of settings made outside the Assignment Editor user interface
- Category, node filter, information, edit bars and spreadsheet
- Integration with other Quartus® II features
- Enhanced spreadsheet interface
- Dynamic Syntax checker
- Node Filter bar
- Using Time Groups
- Customizable columns
- Tool Command Language (Tcl) interface
- Exporting Assignments
- Importing Assignments

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 and minimizable window. This scalability makes it easy to view or edit your assignments right next to your design files. You can open the Assignment Editor by choosing **Assignment Editor** (Assignments menu) or by clicking on the Assignment Editor icon in the toolbar.

Effects of Settings Made Outside the Assignment Editor User Interface

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, the Assignment Editor updates automatically if you add, remove or change an assignment outside the Assignment Editor.

The Assignment Editor is refreshed each time you click anywhere in the window. You can also refresh the Assignment Editor window by selecting **Refresh** (View menu). If you make an assignment in the Quartus® II software, such as in the tool command language (Tcl) console or in the Floorplan Editor, the Assignment Editor reloads the new assignments from memory. If you modify the Quartus II Setting File (.qsf) outside the Assignment Editor, you can choose **Refresh** (View menu) or click on the **Assignment Editor** (Assignments Menu). The Assignment Editor reloads the QSF and displays the latest assignments stored in the QSF file.

-  If the QSF is edited while the project is open, Altera recommends that you choose **Save Project** (File menu) to ensure that you are editing the latest QSF file.

In each case, the Messages window displays the following message:

```
Info: Assignments reloaded -- assignments updated  
outside Assignment Editor
```

The assignments you make in the Assignment Editor, Timing Closure Floorplan, or with Tcl are stored in memory. Choose one of the following commands to write these assignments into the QSF:

- **Close Project** (File menu)
- **Save Project** (File menu)
- **Start Compilation** (Processing menu)

Category, Node Filter, Information, Edit Bars & Spreadsheet

The Assignment Editor window is divided into four bars and a spreadsheet (see [Figure 1–1](#)). 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.

Figure 1–1. The Assignment Editor Window

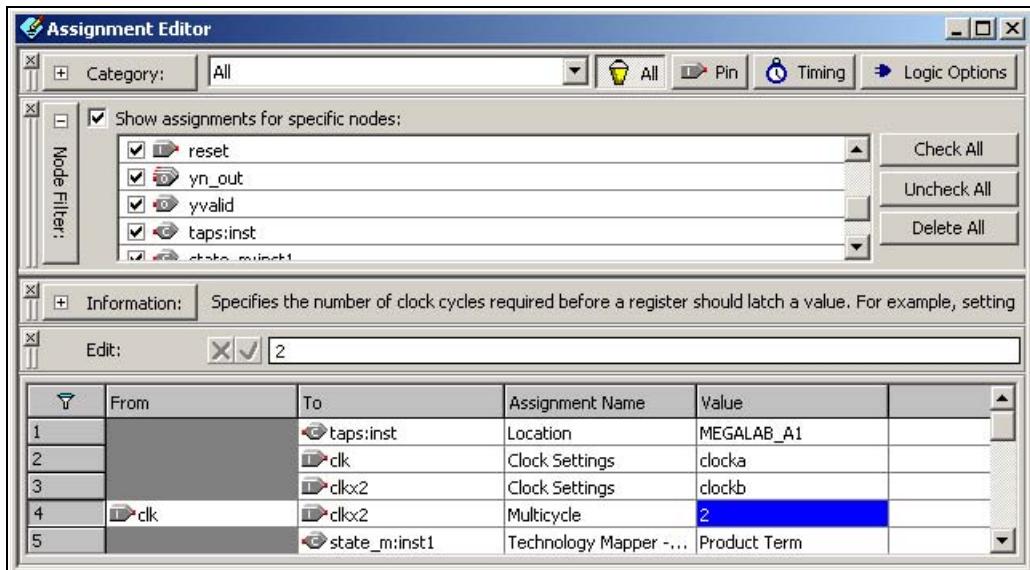


Table 1–1. Assignment Editor Bar Descriptions

Bar Name	Description
Category	Filters the type of available assignments
Node Filter	Filters a selection of design nodes to be viewed or assigned
Information	Displays a description of the cell currently selected
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 chosen 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 t_{SU} in the category list as shown in Figure 1–2. If you select All in the **Category** bar, as shown in Figure 1–3, the Assignment Editor displays all assignments.

Figure 1–2. t_{SU} Selected in the Category List

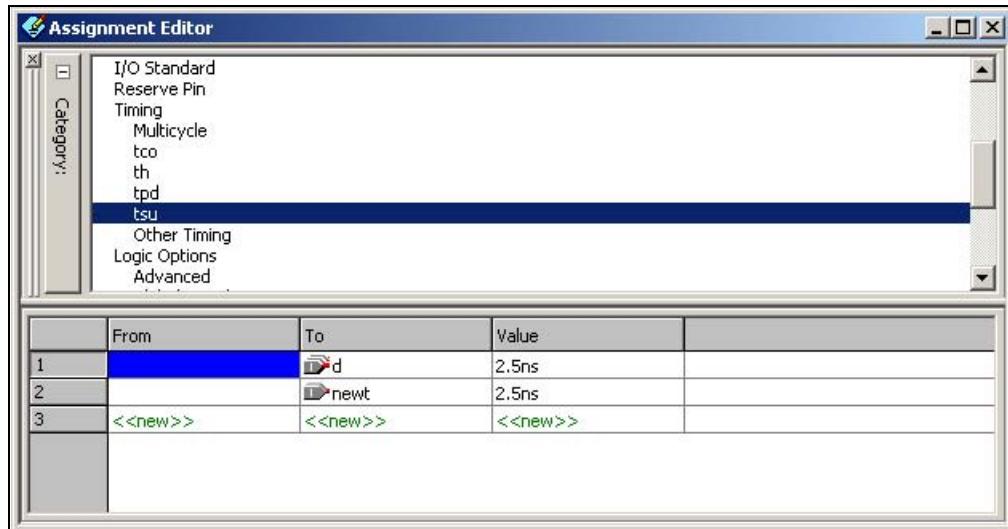
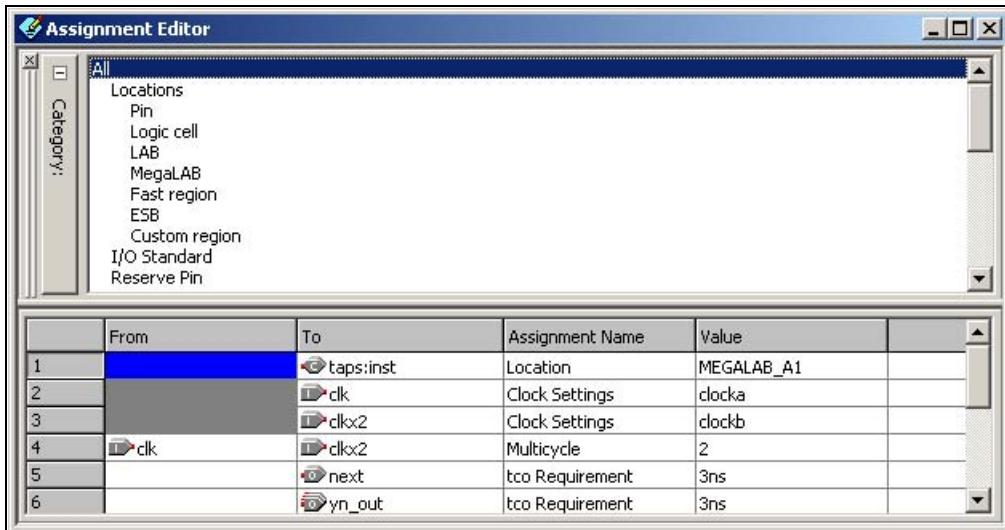


Figure 1–3. All Selected in the Category List

When you collapse the **Category** bar, four shortcut buttons appear allowing you to select from various preset categories (see [Figure 1–4](#)).

Figure 1–4. Category Bar

Node Filter Bar

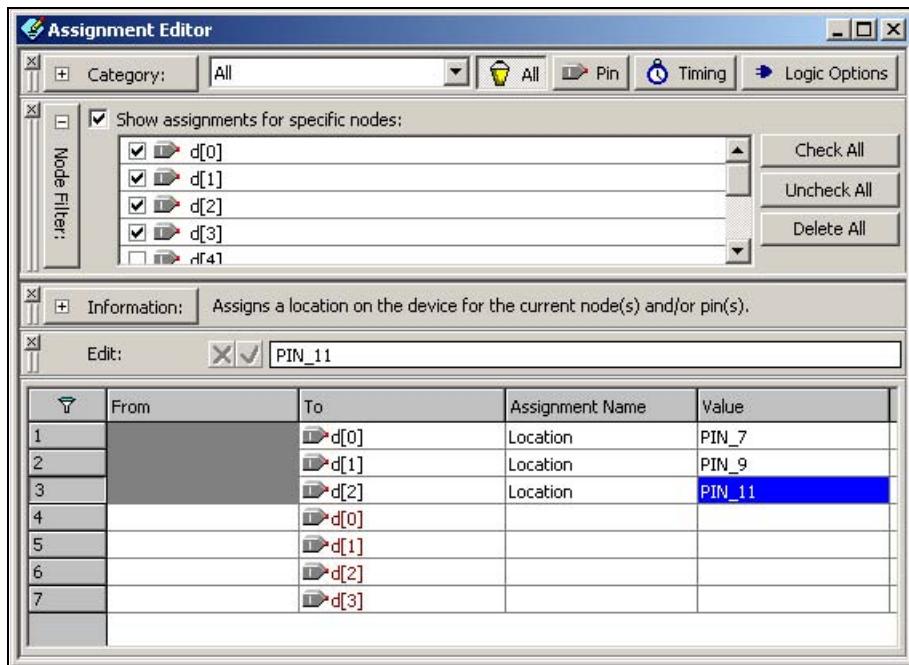
When **Show assignments for specific nodes** is turned on, the spreadsheet shows only assignments for nodes matching the selected node name filters in the **Node Filter** bar. You can selectively enable individual node name filters listed in the **Node Filter** bar. You can create a new node name filter 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 prepopulates 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 appear in dark red. When you choose **Save** (File menu), all incomplete rows are removed and a message issued.

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.



Selecting a d input bus only highlights the row, if you want to enable the bus, you must turn on the check box to enable the bus. This is shown in [Figure 1–5](#).

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. This is a useful way for you to learn how to enter node names and assignments into the spreadsheet. For example, if the selected cell is a particular logic option, the **Information** bar shows a description of that option.



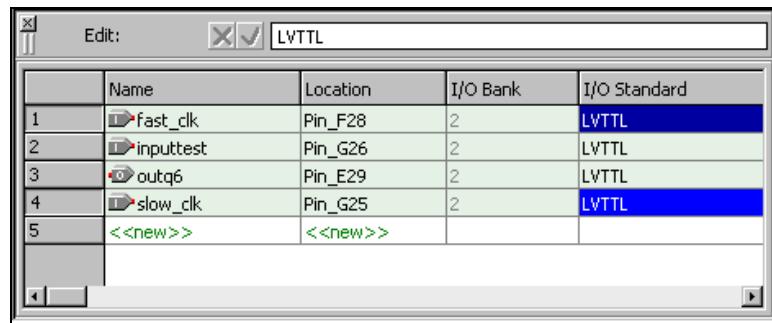
For more information on logic options, see the Quartus II Help.

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 (see [Figure 1–6](#)), then type the new value into the **Edit** box in the **Edit** bar and click the **checkmark** icon (Accept), as shown in [Figure 1–7](#).

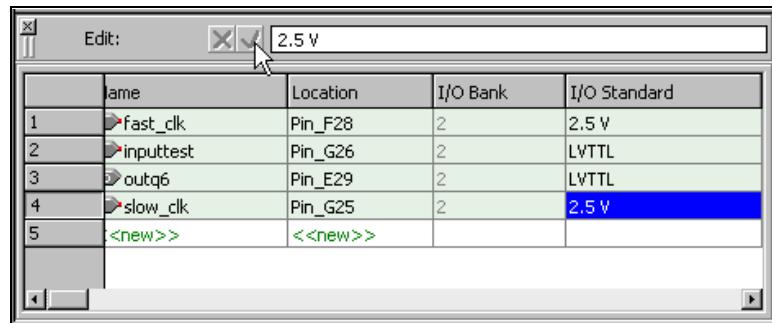
Figure 1–6. Edit Bar Selection



A screenshot of the Assignment Editor software interface. The window title is "Edit". In the top right corner, there is an "Edit" button with a checkmark icon and a text input field containing "LVTTL". Below the toolbar is a spreadsheet table with five rows and five columns. The columns are labeled "Name", "Location", "I/O Bank", and "I/O Standard". Row 1 contains "fast_clk", "Pin_F28", "2", and "LVTTL". Row 2 contains "inputtest", "Pin_G26", "2", and "LVTTL". Row 3 contains "outq6", "Pin_E29", "2", and "LVTTL". Row 4 contains "slow_clk", "Pin_G25", "2", and "LVTTL". Row 5 is a blank row with "<<new>>" in all four columns. The entire row for "slow_clk" is highlighted with a blue selection bar.

	Name	Location	I/O Bank	I/O Standard
1	fast_clk	Pin_F28	2	LVTTL
2	inputtest	Pin_G26	2	LVTTL
3	outq6	Pin_E29	2	LVTTL
4	slow_clk	Pin_G25	2	LVTTL
5	<<new>>	<<new>>		

Figure 1–7. Edit Bar Change



A screenshot of the Assignment Editor software interface, similar to Figure 1–6. The window title is "Edit". In the top right corner, there is an "Edit" button with a checkmark icon and a text input field containing "2.5 V". Below the toolbar is a spreadsheet table with the same structure as Figure 1–6. The "I/O Standard" column for Row 4 ("slow_clk") now contains "2.5 V", indicating the change made in the edit bar. The rest of the table remains the same as in Figure 1–6.

	Name	Location	I/O Bank	I/O Standard
1	fast_clk	Pin_F28	2	2.5 V
2	inputtest	Pin_G26	2	LVTTL
3	outq6	Pin_E29	2	LVTTL
4	slow_clk	Pin_G25	2	2.5 V
5	<<new>>	<<new>>		

Assignment Editor Features

You can open the Assignment Editor from many locations, including the Text Editor, the Node Finder, the Timing Closure Floorplan, 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: Timing Closure Floorplan, Chip Editor, Block Editor, and 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 entry boxes, and copy and paste multiple cells in the Assignment Editor. As you enter an assignment, the font color of the row changes to indicate the status of the assignment.

See “[Dynamic Syntax Checking](#)” 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 appears beside each node name and node name filter to identify its type. The node type icon identifies the entry as an input, output, or bidirectional pin, a register, or combinational logic, see [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

	From	To	Assignment Name
1		enable	Location
2		time[0]	Location
3		auto_max:auto ~406	Location

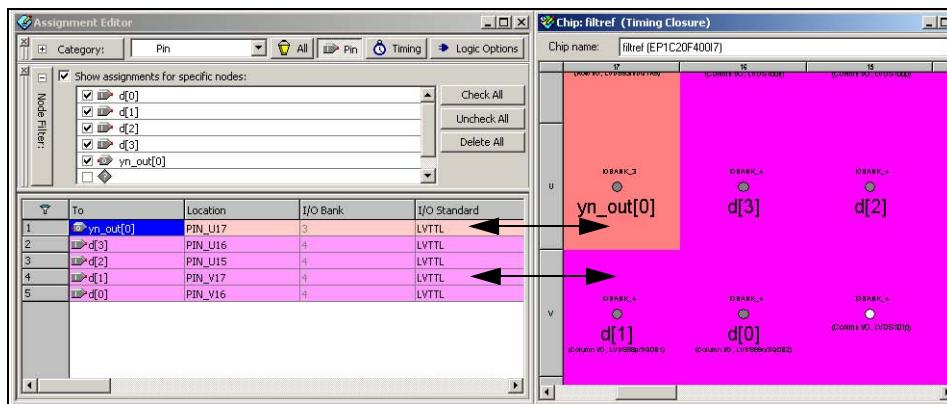
The Assignment Editor supports wildcards in the following types of assignments:

- 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
- LogicLock™ region assignments

The spreadsheet also supports customizable columns, (see “[Customizable Columns](#)” on page 1-12), that allow you to show, hide, and arrange the columns.

When making pin location assignments, the background color of the cells coordinates with the color of the I/O bank also shown in the Floorplan Editor (see [Figure 1-9](#)).

Figure 1-9. Spreadsheet-Like Interface



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 catches general incorrect settings. For example, the Assignment Editor does not allow assignment of a pin 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 (see [Table 1–2](#)). You can customize the colors in the **Options** dialog box (Tools menu).

Table 1–2. Description of the Text Color in the Spreadsheet

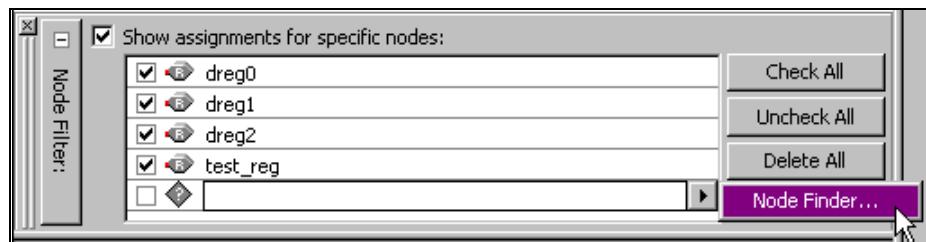
Text Color	Description
Green	A new assignment can be created
Yellow	The assignment contains warnings, such as an unknown node name
Dark Red	The assignment is incomplete
Bright Red	The assignment has an error, such as an illegal value
Light Gray	The assignment is disabled

Node Filter Bar

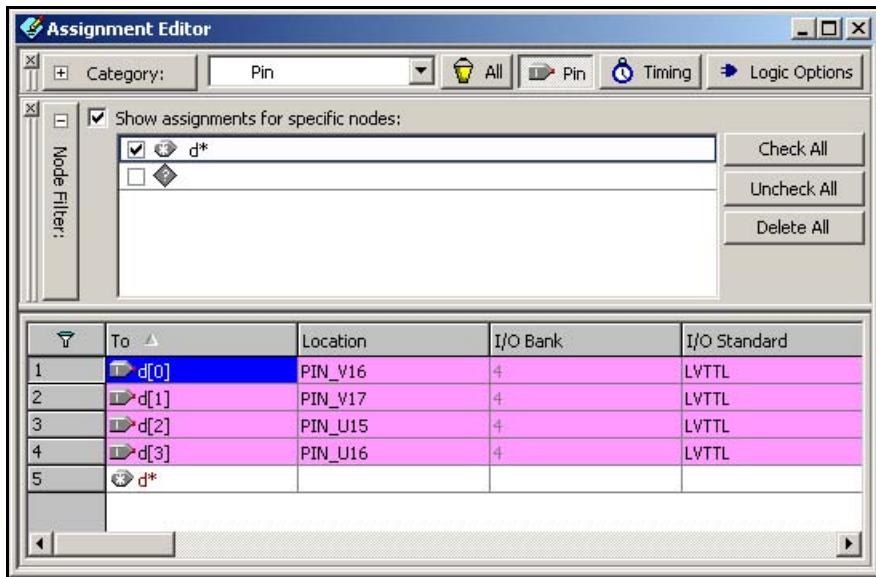
The **Node Filter** bar provides flexibility in how you view and make your settings. The **Node Filter** bar contains a list of node filters. To create a new entry, use the **Node Finder** or manually type the node name.

Double-click an empty row in the **Node Filter** list and then click on the arrow to open the **Node Finder** (see [Figure 1–10](#)).

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 (?) can be 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] (see [Figure 1–11](#)).

Figure 1–11. Using the Node Filter Bar with Wildcards

Using 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. Using time groups with the Assignment Editor provides the flexibility required for making complex timing assignments to a large number of nodes.

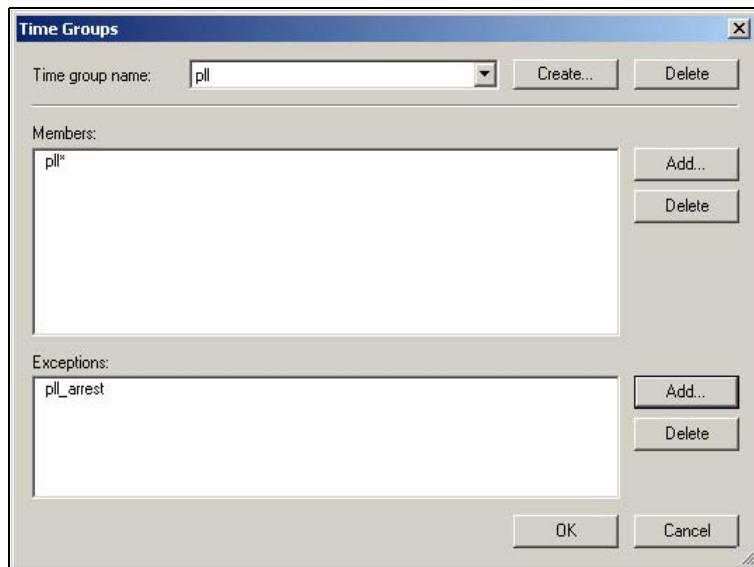
To create a time group, open the **Time Groups** dialog box by choosing **Time Groups** (Assignments menu). You can add or delete members of each time group with wild cards in the **Node Finder** (see [Figure 1–12](#)).



For more information on using Time Groups for timing analysis, see the *Quartus II Timing Analysis* chapter in Volume 3 of the *Quartus II Handbook*.

There are cases when wild cards are not flexible enough to select a large number of nodes that have node names that are quite similar. With time groups you can combine wild cards, which select a large number of nodes, and use exceptions to remove nodes that you did not intend to select.

Figure 1–12. Time Groups Dialog Box



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, as well as align (left, center, or right) the content in the column 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 **Comments** column is hidden. However, you can show or hide any of the available columns by choosing the **Customize Columns** command (View menu). When you restart the Quartus II software, the column settings are maintained.

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.

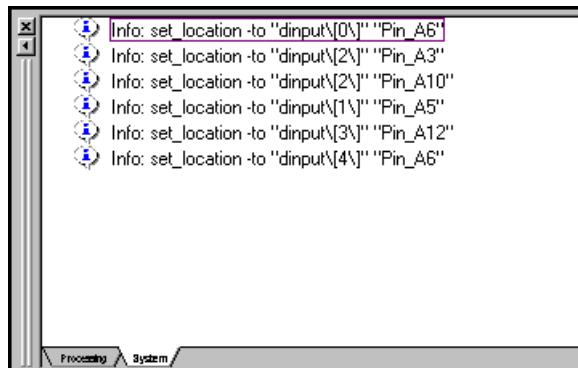
Tcl Interface

Whether you use the Assignment Editor or another feature to create your design's assignments, you can export them all to a Tcl file. You can then use the Tcl file to reapply all the settings or to archive your assignments. Choose **Export** (File menu) to export your assignments to a Tcl script.

 You can also choose the **Generate Tcl File for Project** (Project menu) to generate a Tcl script file for your project.

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 (see [Figure 1–13](#)). To copy a Tcl command from the **Messages** window, right-click the message and choose **Copy** (right button pop-up menu).

Figure 1–13. Equivalent Tcl Commands Displayed in the Messages Window



 For more information on Tcl scripting with the Quartus II software, see the *Tcl Scripting* chapter in Volume 2 of the *Quartus II Handbook*.

Creating Pin Locations Using the Assignment Editor

Altera recommends using one of two methods for creating pin assignments. The first approach involves choosing a pin, from your design, for each device pin location. The second approach involves choosing a pin location for each pin in your design.

Setting Pin Locations From the Pin Number List

It is important to understand the properties of the pin before assigning a pin name to that location. For example, you will need to know which I/O bank or VREF group the pin number belongs to when following pin placement guidelines.



For more information on pin placement guidelines, see the *Selectable I/O Standards* chapters in the appropriate device handbook.

Before creating pin assignments, analysis and elaboration should be performed on your design to create a database of your design pin names.

1. Choose **Assignment Editor** (Assignments menu).

 You can also launch the Assignment Editor by pressing **Ctrl+Shift+A**.

2. Select **Pin** in the **Category Bar**.

Creating pin assignments can be difficult when you need to check which I/O bank the pin number is in, or which VREF pad the pin uses. By selecting the **Pin** category, more pin related information (I/O bank, VREF pad, I/O Standard, pin functionality, and so on) is visible in the spreadsheet to help you create pin location assignments.

3. Choose **Show All Assignable Pin Numbers** (View menu).

A list of all assignable pin numbers (user I/O pins) for the targeted device appears under the **Location** column.

4. Choose **Customize Columns** (View menu) to add and remove columns to the spreadsheet in the Assignment Editor.

There are many columns that are not shown by default, including toggle rate, timing requirement, and fast input and output register options.

5. Find a pin number in the spreadsheet and double-click the cell next to the pin location under the **To** column. Type the pin name or choose a pin from the drop-down list. A drop-down list of all your design pins appears if an analysis has been performed.



As you type in a pin name, the Assignment Editor automatically completes the field by referencing the pin names stored in the database created from the initial analysis and elaboration. Also, pin names already assigned to a pin location appear in italics.

Setting Pin Locations From the Design Pin Name List

It is important to understand the properties of the pin before assigning a pin name to that location. For example, you need to know which I/O bank or VREF group the pin number belongs to when following pin placement guidelines.



For more information on pin placement guidelines, see the *Selectable I/O Standards* chapters in the appropriate device handbook.

To select pin name and assign to a location follow these steps:

1. Choose **Assignment Editor** (Assignments menu).

 You can also launch the Assignment Editor by pressing **Ctrl+Shift+A**.

2. Select **Pin** from the Category Bar.

Creating pin assignments is difficult when you need to check which I/O bank the pin number is in, or which VREF pad the pin uses. By selecting the **Pin** category, more pin related information (I/O bank, VREF pad, I/O Standard, Pin Functionality, and so on) is visible in the spreadsheet to help you create pin location assignments.

3. Choose **Show All Known Pin Names** (View menu).

A list of all assignable pin names in your design appears under the **To** column.

4. Choose **Customize Columns** (View menu) to add and remove additional columns to the spreadsheet in the Assignment Editor.

There are many columns that are not shown by default, including toggle rate, timing requirements and fast input/output register options.

5. Find a pin name in the spreadsheet and double-click the cell next to the pin name under the **Location** column. Select a pin number from the drop-down list containing all assignable pin numbers in the targeted device. You can also type in the pin number and let the Assignment Editor automatically complete the pin number.



Instead of typing `Pin_AA3`, you can type `AA3` and let the Assignment Editor auto complete the pin number to `Pin_AA3`.



To learn more about efficiently creating pin assignments with the Assignment Editor, see the *I/O Assignment Planning and Analysis* chapter in Volume 2 of the *Quartus II Handbook*.

Exporting & Importing Assignments

Designs that use the LogicLock hierachal design methodology uses 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.

With the **Export Assignments** and **Import Assignments** dialog boxes (Assignments menu), you can export your Quartus II assignments to a QSF, and import assignments from a QSF, a Quartus II Entity Settings File (.esf), a MAX+PLUS® II Assignment and Configuration File (.acf), or a Comma Separated Value File (.csv).

In addition to the **Export Assignments** and **Import Assignments** dialog boxes, the **Export** command (File menu) allows you to export your assignments to a Tcl Script File (.tcl).

 The **Export** command (File menu) exports the contents of the active window in the Quartus II software to another file format, when applicable.

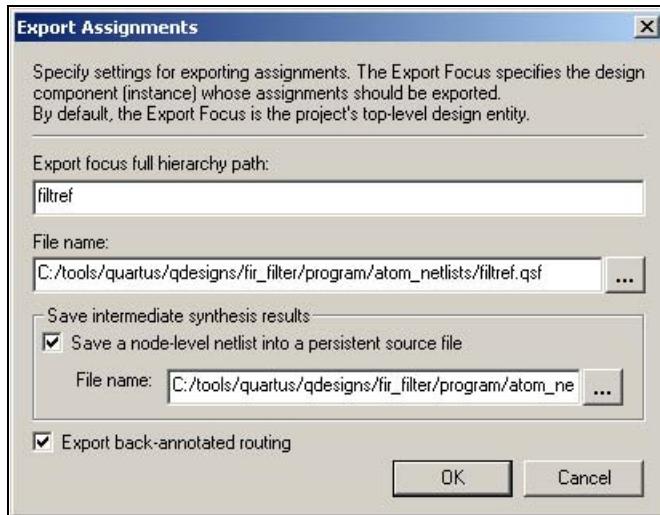
You can use these file formats for many different aspects of your project. For example, you can use a CSV file for documentation purposes or to transfer pin-related information to board layout tools. The Tcl file makes it easy to apply assignments in a scripted design flow. And the LogicLock design flow uses the QSF 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 QSF file, generate a node-level netlist file, and export back-annotated routing information as a Routing Constraints File (.rcf), as shown in Figure 1-14. Choose **Export Assignments** (Assignments menu) to open the **Export Assignments** dialog box. The LogicLock design flow also uses this dialog box to export LogicLock regions.



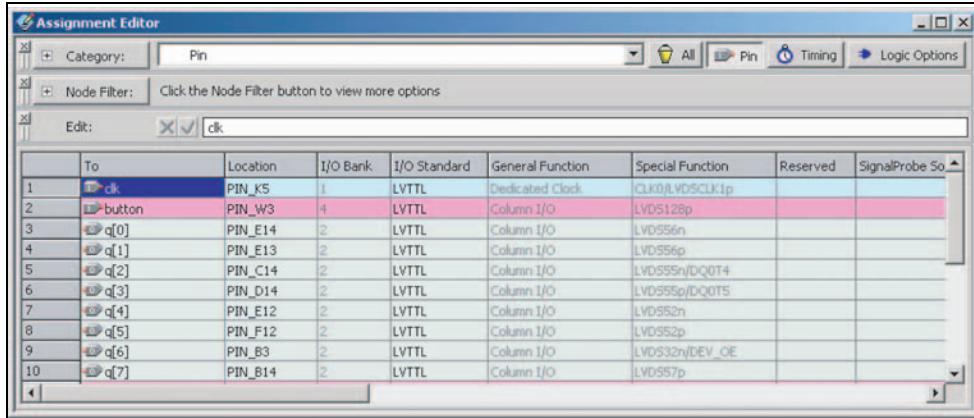
For more information on using the **Export Assignments** dialog box to export LogicLock regions, see the *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*.

Figure 1–14. Export Assignments Dialog Box

You can use the **Export** command (File menu) to export all assignments to a Tcl file or export a set of assignments to a CSV file. When you export assignments to a Tcl file, only user-created assignments are written to the Tcl script file, and default assignments are not exported.

When assignments are exported to a CSV file, only the assignments displayed in the current view of the Assignment Editor are exported. For example, to export only pin assignments, select **Pin** from the **Category** bar. Then, choose **Export** (File menu), and select **Comma Separated Value File** in the **Save as type** list.

The first uncommented row of the CSV 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 (see [Figure 1–15](#)). You can view and make edits to the CSV file with Excel or other spreadsheet tools.

Figure 1–15. Assignment Editor With Category set to Pin

Here is an example of an exported CSV 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 ,SignalProbe Source
clk,PIN_K5,1,LVTTL,Dedicated Clock,CLK0/LVDSCLK1p,,,
button,PIN_W3,4,LVTTL,Column I/O,LVD5128p,,,
q[0],PIN_E14,2,LVTTL,Column I/O,LVD556n,,,
q[1],PIN_E13,2,LVTTL,Column I/O,LVD556p,,,
q[2],PIN_C14,2,LVTTL,Column I/O,LVD555n/DQ0T4,,,
q[3],PIN_D14,2,LVTTL,Column I/O,LVD555p/DQ0T5,,,
q[4],PIN_E12,2,LVTTL,Column I/O,LVD552n,,,
```

Importing Assignments

The **Import Assignments** dialog box allows you to import Quartus II assignments from a QSF, ESF, ACF, or CSV file (see [Figure 1–16](#)). To import assignments from any of the supported assignment files, follow these steps:

1. Choose **Import Assignments** (Assignments menu).
2. In the **File name** text-entry box, enter the file name, or click **Browse (...)** to navigate to the assignment file.
3. When the **Select File** dialog box opens, select the file, and click **Open**.
4. Click **OK** in the **Import Assignments** dialog box.



When you import a CSV 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, follow these steps:

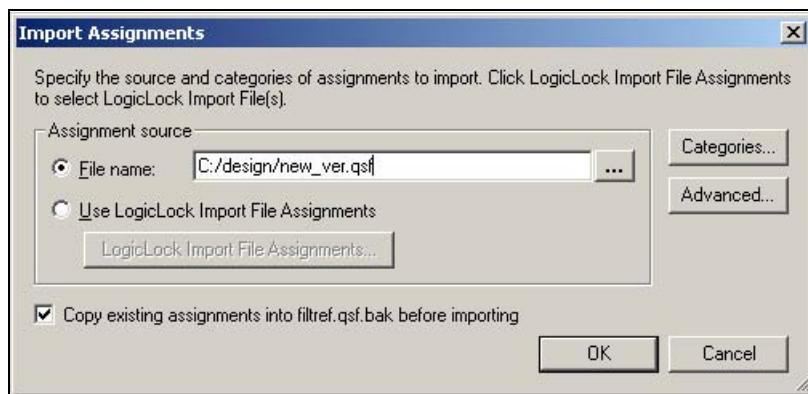
1. Choose **Import Assignments** (Assignments menu).
2. Turn on **Use LogicLock assignments** and click **LogicLock Import File Assignments**.
3. When the **LogicLock Import File Assignments** dialog box opens, select the assignments to import and click **OK**.



For more information on using the **Import Assignments** dialog box to import LogicLock regions, see 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.

Figure 1–16. Import Assignments Dialog Box



When importing assignments from a file, you can choose which assignment categories to import by following these steps:

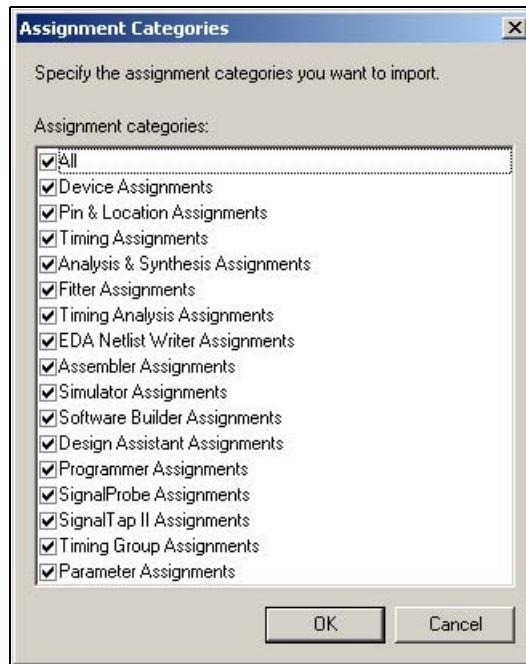
1. Click **Categories** in the **Import Assignments** dialog box.
2. Turn on the checkbox for each of the **Assignment Categories** you want to import, as shown in [Figure 1–17](#).

To select specific types of assignments to import, click **Advanced** in the **Import Assignments** dialog box. The **Advanced Import Settings** dialog box appears and you can choose to import instance, entity, or global assignments, as well as select various assignment types to import.



For more information on these options, see 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**, **Edit**, and **Node Filter** bars, the Assignment Editor provides an efficient assignment entry solution for FPGA designers.

qii52002-2.1

Introduction

FPGA design software that easily integrates into a design flow saves time and improves productivity. The Altera® Quartus® II software provides designers 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, and 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 their use in scripts automating segments of the FPGA design flow.

The Benefits of Command-Line Executables

The Quartus II command-line executables reduce the amount of memory required during any step in the design flow. Because it targets only one step in the design flow, each executable is relatively compact, both in file size and the amount of memory used when running. This memory reduction improves performance for all designers and is particularly beneficial in design environments with heavily used computer networks or workstations with low amounts of memory.

Command-line executables also provide command-line control over each step of the design flow. Each executable has 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. The Quartus II command-line executables can also be integrated 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 for designers who want it 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. As an 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.

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/tutorial` directory.

Before making changes, copy the tutorial directory and type the following four commands at a command prompt in the new project directory.

 The `<Quartus II directory>/bin` directory must be in your PATH environment variable.

```
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_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. The 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_tan filtref` command performs timing analysis on the **filtref** project to determine whether the design meets the timing requirements that were specified by the `quartus_fit` executable.

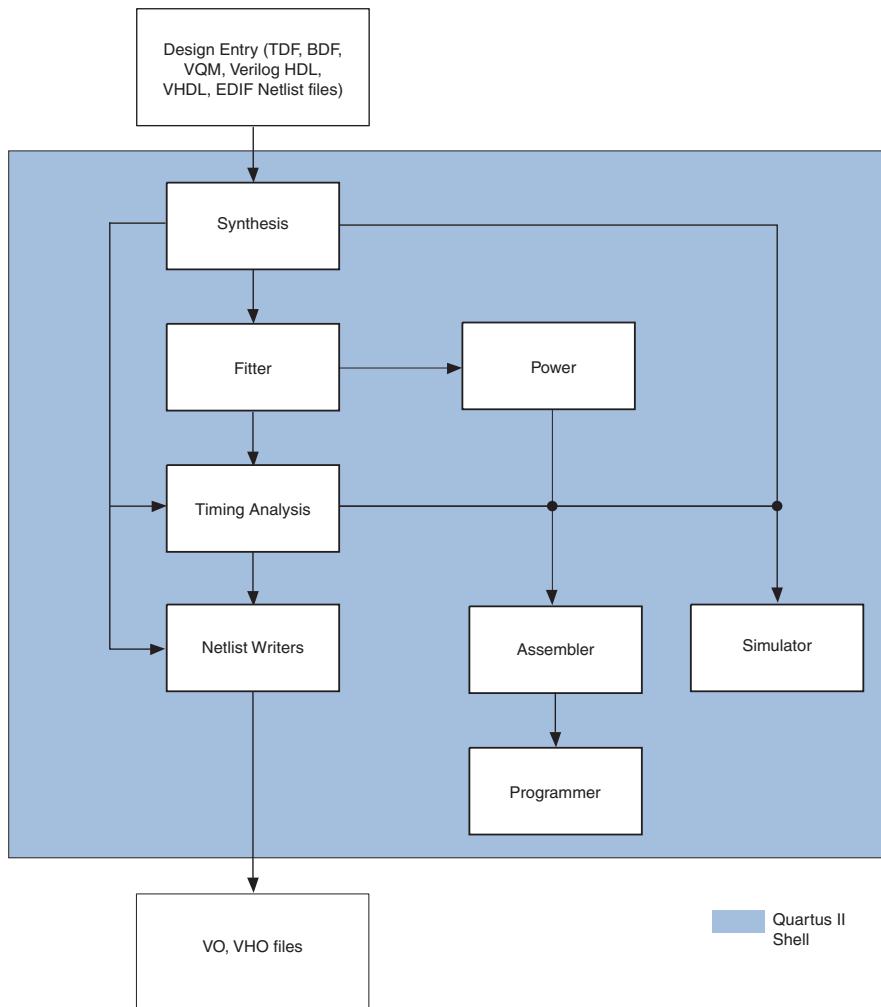
The `quartus_asm filtref -` command creates programming files for the **filtref** project.

These four commands are stored in a batch file for use on PCs or in a shell script file for use on UNIX workstations.

Design Flow

Figure 2–1 shows a typical design flow.

Figure 2–1. Typical Design Flow



Command-line executables are provided for each stage in the design flow shown in [Figure 2–1](#). Additional command-line executables are provided for specific tasks. [Table 2–1](#) lists each Quartus II command-line executable and provides a brief description of its function.

Table 2–1. Quartus II Command-line Executables & Descriptions (Part 1 of 3)

Executable	Description
Analysis & Synthesis quartus_map	Quartus II Analysis & 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 & Synthesis must be run successfully before running the Fitter.
Timing Analyzer quartus_tan	The Quartus II Timing Analyzer computes delays for the given design and device, and annotates them on the netlist. Then, the Timing Analyzer performs timing analysis, allowing you to analyze the performance of all logic in your design. The quartus_tan executable includes tool language support (Tcl) support. Quartus II Analysis & Synthesis or the Fitter must be run successfully before running the Timing Analyzer.
Assembler quartus_asm	The Quartus II Assembler generates a device programming image, in the form of one or more Programmer Object Files (.pof), SRAM Object Files (.sof), Hexadecimal (Intel-Format) Output Files (.hexout), Tabular Text Files (.ttf), and Raw Binary Files(.rbf), from a successful fit (that is, place-and-route). 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 .hexout.tff , TTFs, and RBFs 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.
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 & Synthesis or the Fitter must be run successfully before running the Design Assistant.

Table 2–1. Quartus II Command-line Executables & Descriptions (Part 2 of 3)

Executable	Description
Compiler Database Interface quartus_cdb	<p>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.</p> <p>Analysis & Synthesis must be run successfully before running the Compiler Database Interface.</p>
EDA Netlist Writer quartus_eda	<p>The Quartus II EDA Netlist Writer generates netlist and other output files for use with other EDA tools.</p> <p>Analysis & Synthesis, the Fitter, or Timing Analyzer must be run successfully before running the EDA Netlist Writer, depending on the arguments used.</p>
Simulator quartus_sim	<p>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.</p> <p>Quartus II Analysis & Synthesis must be run successfully before running a functional simulation.</p> <p>The Timing Analyzer must be run successfully before running a timing simulation.</p>
Software Build quartus_swb	<p>The Quartus II Software Builder performs a software build, which processes a design for an ARM®-based Excalibur™ device or the Nios® embedded processor.</p>
Programmer quartus_pgm	<p>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), or Jam Byte-Code File (.jbc).</p> <p>Make sure you specify a valid programming mode, programming cable, and operation for a specified device.</p>
Convert Programming File quartus_cpf	<p>The Quartus II Convert Programming File module converts one programming file format to a different possible format.</p> <p>Make sure you specify valid options and an input programming file to generate the new requested programming file format.</p>

Table 2–1. Quartus II Command-line Executables & Descriptions (Part 3 of 3)	
Executable	Description
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: quartus_tan , quartus_cdb , and quartus_sim . 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.
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 & Synthesis or the Fitter must be run successfully before running the PowerPlay Power Analyzer.

Text-Based Report Files

Each command-line executable creates a text-format report file when it is run. These files report success or failure, and contain information on 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 on this package, see “[More Help with Quartus II Command-Line Executables](#)” on page 2–20.

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 on 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, Altera recommends that you use the **quartus_sh --flow** option in the Quartus II software version 4.1.

The following example runs compilation, timing analysis, and programming file generation—with a single command.

```
quartus_sh --flow compile filtref ↵
```

Command-Line Scripting Help

Complete help information is integrated with each command-line executable. Access help for a executable using the **-h** option. For example, to view help for the **quartus_map** executable, run the command **quartus_map -h**. The following example shows the result of running **quartus_map -h**:

```
C:\>quartus_map -h
Quartus II Analysis & Synthesis
Version 4.1 Internal Build 133 04/07/2004 SJ Full Version
Copyright (C) 1991-2004 Altera Corporation

Usage:
-----
quartus_map [-h | --help[=<option|topic>] | -v]
quartus_map <project name> [<options>]

Description:
-----
Quartus(R) II Analysis & 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.

Options:
-----
-f <argument file>
-c <revision name> | --rev=<revision name>
```

```

-l <path> | --lib_path=<path>
--lower_priority
--optimize=<area|speed|balanced>
--family=<device family>
--part=<device>
--
state_machine_encoding=<auto|minimal_bits|one_hot|user_enco
--enable_register_retimining[=on|off]
--enable_wysiwyg_resynthesis[=on|off]
--ignore_carry_buffers[=on|off]
--ignore_cascade_buffers[=on|off]
--analyze_project
--analyze_file=<design file>
--generate_symbol=<design file>
--generate_inc_file=<design file>
--convert_bdf_to_verilog=<.bdf file>
--convert_bdf_to_vhdl=<.bdf file>
--export_settings_files[=on|off]
--generate_functional_sim_netlist
--source=<source file>
--update_wysiwyg_parameters

Help Topics:
-----
arguments
makefiles

```

For more information on specific options, use --help=<option|topic>.

Detailed help about a particular option is also available. For example, to view detailed help about the --optimize option, run quartus_map --help=optimize. The following is the result of running quartus_map --help=optimize:

Option: --optimize=<area|speed|balanced>

Option to optimize the design to achieve maximum speed performance, minimum area usage, or high speed performance with minimal area cost during synthesis.

The following table displays available values:

Value	Description
area	Makes the design as small as possible in order to minimize resource usage.
speed	Chooses a design implementation that has the fastest fmax.
balanced	Chooses a design implementation that has a high-speed performance with minimal logic usage Note that the current version of the Quartus(R) II software does not support the "balanced" setting for the following devices:

Mercury(TM), MAX(R) 7000B/7000AE/3000A/7000S/7000A,
FLEX(R) 6000, FLEX 10K(R), FLEX 10KE/10KA, and ACEX 1K.



Help on Quartus II command-line executables is also available by typing `quartus_sh --qhelp` at a command prompt. For more information, see “[More Help with Quartus II Command-Line Executables](#)” on page 2–20.

Command-Line Option Details

Command-line options are provided for making 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. See “[Option Precedence](#)” on page 2–9 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, you should choose this method.

Option Precedence

If you are using the command-line executables, you need to 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 (.qsf) for the project. Assignments for a project can also be made by using 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 QSF or the compiler database files. There are two command-line options to specify whether QSF 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 QSF or compiler database files is set to its default value.

The file precedence command-line options are `--import_settings_files` and `--export_settings_files`. By default, the `--import_settings_files` and `--export_settings_files` options are turned on. Turning on the `--import_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

--export_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 --import_settings option.

Table 2–2. Precedence for Reading Assignments	
Option Specified	Precedence for Reading Assignments
--import_settings_files=on (Default)	1. Command-line options 2. Quartus II Settings File (.qsf) 3. Project database (db directory, if it exists) 4. Quartus II software defaults
--import_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 --export_settings command-line option.

Table 2–3. Location for Writing Assignments	
Option Specified	Location for Writing Assignments
--export_settings_files=on (Default)	Quartus II Settings File (.qsf) and compiler database
--export_settings_files=off	Compiler database

The following example assumes that a project named **fir_filter** exists, and that the analysis and synthesis step has been performed (using the **quartus_map** executable).

```
quartus_fit fir_filter --fmax=80MHz ←
quartus_tan fir_filter ←
quartus_tan fir_filter --fmax=100MHz --tao=timing_result-100.tao ←
--export_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 `--export_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 `--export_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 `--import_settings_files=off` and `--export_settings_files=off` options (where appropriate) to optimize the way that the Quartus II software reads and updates settings files. The following example shows how to avoid unnecessary importing and exporting.

```
quartus_map filtref --source=filtref
    --part=ep1s10f780c5 ←
quartus_fit filtref --fmax=100MHz
    --import_settings_files=off ←
quartus_tan filtref --import_settings_files=off
    --export_settings_files=off ←
quartus_asm filtref --import_settings_files=off
    --export_settings_files=off ←
```

The **quartus_tan** and **quartus_asm** executables do not need to import or export settings files because they do not change any settings in the project.

Command-Line Scripting Examples

This section of the chapter presents various examples of command-line executable use.

Create a Project & 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 with the **Generate Tcl File for Project** command (Project menu) in the Quartus II GUI.

The following Tcl script example creates a project and applies project constraints using the tutorial design files in the *<Quartus II installation directory>/qdesigns/tutorial/* directory.

```
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 use the following commands 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.

```
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 following shell script example 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.

The following shell script is specifically designed for use on UNIX systems employing the sh shell.

```
#!/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 & 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_retimining=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 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 following command at a command prompt:

```
quartus_map top --source=top.edf --enable_register_retimining=on
--enable_wysiwyg_resynthesis=on --part=EP20K600EBC652-1X
```

Attempt to 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 forces 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 the following command at a command prompt:

```
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 a file called **fir_filter.qpf**). If the tutorial files are installed on your system, this project exists in the *<Quartus II directory>/qdesigns/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.

This shell script is specifically designed for use on UNIX systems employing the **sh** shell.

```
#!/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 on the DSE, type `quartus_sh --help=dse ↵` at the command prompt, or see the *Design Space Explorer* chapter in Volume 2 of the *Quartus II Handbook*.

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 all assignments for a Quartus II project are stored in one file, the Quartus II Settings File (.qsf), including the QSF 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 QSF, requiring a complete recompilation if the QSF 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 QSF, 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 QSF in each makefile rule, include only the appropriate change file.

The following example 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 ↵`.

```
#####
# Project Configuration:
#
# Specify the name of the design (project) 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
# clean_all: removes settings files as well as clean.
#####
```

```

all: smart.log $(PROJECT).asm.rpt $(PROJECT).tan.rpt

clean:
    rm -rf *.rpt *.chg smart.log *.htm *.eqn *.pin *.sof *.pof db
    rm -rf *.summary
clean_all: clean
    rm -rf *.qpf*.qsf *.qws

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 command `quartus_sh --help=determine_smart_action ↵`.

The QFlow Script

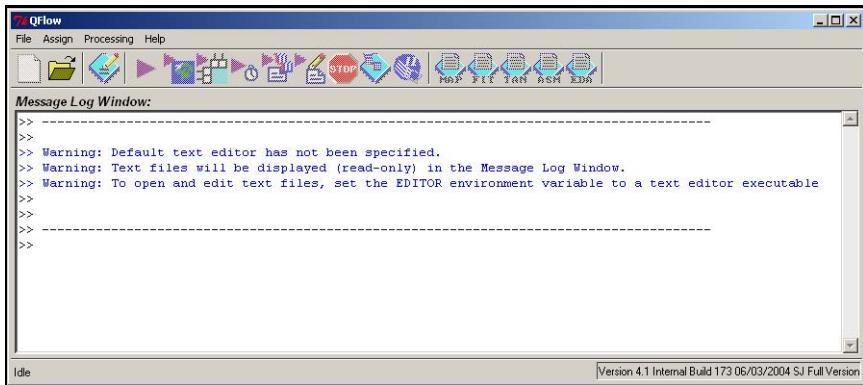
A Tcl/Tk-based graphical interface called QFlow is included with the command-line executables. Designers can use the QFlow interface to open projects, launch some of the command-line executables, view report files, and make some global project assignments. The QFlow interface can run the following command-line executables:

- **quartus_map** (Analysis & 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 GUI.

Start QFlow by typing the following command at a command prompt:
`quartus_sh -g ↵`. [Figure 2-2](#) shows the QFlow interface.

Figure 2–2. QFlow Interface

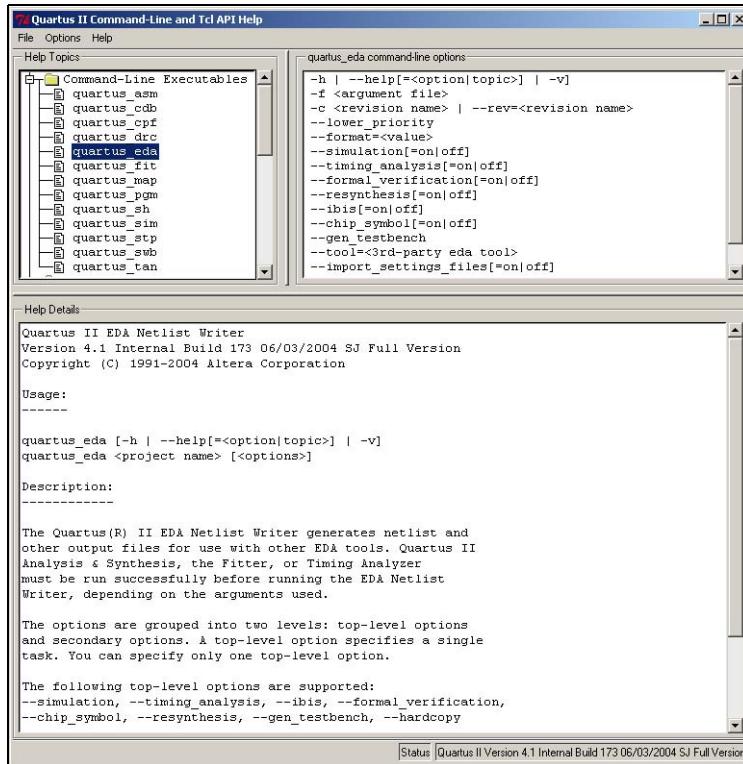


The QFlow script is located in the
<Quartus II directory>/bin/tcl_scripts/qflow/ directory.

More Help with Quartus II Command-Line Executables

More information on command-line executable use and the Quartus II Tcl API is available by typing `quartus_sh --qhelp ↵` at a command prompt. This command starts the Quartus II Command Line and Tcl API Help browser, a viewer for information on the Quartus II command-line executables and Tcl API (Figure 2–3).

Figure 2–3. Quartus II Command Line & Tcl API Help Browser



Click items under **Help Topics** to get more information on the topics listed.

Conclusion

Command-line scripting in Quartus II software provides important benefits to designers, including increased flexibility and easy integration with other EDA software in FPGA design flows. Scripts reduce memory usage, improve performance, and bring true command-line control to all stages of FPGA design.

qii52003-2.2

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 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 migrating assignments to another project. 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.

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.

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, see “[References](#)” on page 3–34.

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.

The Quartus II software, beginning with version 4.1, supports Tcl/Tk version 8.4, supplied by the Tcl DeveloperXchange at tcl.activestate.com.

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 `worl`d 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 3–3. 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. The following example assigns the value 1 to the variable named `a`:

```
set a 1
```

To access the contents of a variable, use a dollar sign before the variable name. The following example prints "Hello world" in a different way:

```
set a Hello
set b world
puts "$a $b"
```

Substitutions

Tcl performs three types of substitution: variable value substitution, nested command substitution, and backslash substitution.

Variable Value Substitution

Variable Value substitution, in the previous example, 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. This example sets `a` to the length of the string `foo`:

```
set a [string length foo]
```

Backslash Substitution

Backslash substitution allows you to quote reserved characters in Tcl, like 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. This example shows how to use the backslash character for line continuation:

```
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. This example sets `b` to the sum of the value in the variable `a` and the square root of 2:

```
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. This example sets `a` to a list with three numbers in it:

```
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. This example prints the second element (at index 1) in the list stored in `a`:

```
puts [lindex $a 1]
```

The `llength` command returns the length of a list. This example prints the length of the list stored in `a`:

```
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. This example appends some elements to the list stored in `a`:

```
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 }
```

The following example shows how to access the value for a particular index:

```
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. The following example shows one way to iterate over all the values in an array:

```
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. Positioning curly braces as shown in the following examples ensures the control structure commands are executed efficiently and correctly. This example prints whether the value of variable a is positive, negative, or zero:

```
if { $a > 0 } {
    puts "The value is positive"
} elseif { $a < 0 } {
    puts "The value is negative"
} else {
    puts "The value is zero"
```

```
}
```

This example uses a `for` loop to print each element in a list:

```
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]"
}
```

This example uses a `foreach` loop to print each element in a list:

```
set a { 1 2 3 }
foreach element $a {
    puts "The list element is $element"
}
```

This example uses a `while` loop to print each element in a list:

```
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. This example defines a procedure that multiplies two numbers and returns the result:

```
proc multiply { x y } {
    set product [expr { $x * $y }]
    return $product
}
```

This example shows how to use the `multiply` procedure in your code. You must define a procedure before your script calls it, as shown in this example:

```
proc multiply { x y } {
    set product [expr { $x * $y }]
```

```

        return $product
    }
    set a 1
    set b 2
    puts [multiply $a $b]
}

```

Altera recommends defining 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. This example defines a procedure that prints an element in a global list of numbers, then calls the procedure:

```

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 this example:

```
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 filehandle 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 this example:

```
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. This is a simple example with the `gets` command that prints out each line of the file, with its line number:

```
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 & 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 3–3, 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 command. 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. The following example is a valid line of code that includes a set command and a comment:

```
set a 1;# Initializes a
```

Without the semicolon, it will be an invalid command because the set command will not terminate until the newline after the comment.

The Tcl interpreter counts curly braces inside comments, which can lead to errors that are difficult to track down. The following example will cause an error because of unbalanced curly braces:

```
# if { $x > 0 } {
if { $y > 0 } {
    # code here
}
```

Quartus II Tcl API Help

Access the Quartus II Tcl API Help reference by typing the following command at a command prompt:

```
quartus_sh --qhelp
```

This command runs the Quartus II Command-Line and the 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.



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.

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 Commands Grouped in Packages, by Function

Package Name	Package Description
project	Create and manage projects and revisions, make any project assignments including timing assignments
flow	Compile a project, run command-line executables and other common flows
report	Get information from report tables, create custom reports
timing	Annotate timing netlist with delay information, compute and report timing paths
timing_report	List timing paths
advanced_timing	Traverse the timing netlist and get information about timing nodes
device	Get device and family information from the device database
backannotate	Back annotate assignments
logiclock	Create and manage LogicLock regions
chip_editor	Identify and modify resource usage and routing with the Chip Editor
simulator	Configure and perform simulations
stp	Run the SignalTap® II logic analyzer
database_manager	Manage version-compatible database files
misc	Perform miscellaneous tasks

By default, only the minimum number of packages are 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, or get help on 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 blank space means the package is not available in that executable.

Table 3–2. Tcl Package Availability by Quartus II Executable					
Packages	Quartus II Executable				
	Quartus_sh	Quartus_tan	Quartus_cdb	Quartus_sim	Tcl Console
advanced_timing		Not Loaded			
backannotate			Not Loaded		Not Loaded
chip_editor			Not Loaded		
device	Loaded	Not Loaded	Loaded	Loaded	Not Loaded
flow	Not Loaded	Not Loaded	Not Loaded	Not Loaded	Not Loaded
logiclock		Not Loaded	Not Loaded		Not Loaded
misc	Loaded	Loaded	Loaded	Loaded	Loaded
project	Loaded	Loaded	Loaded	Loaded	Loaded
report	Not Loaded	Not Loaded	Not Loaded	Loaded	Not Loaded
simulator				Loaded	
timing		Loaded			
timing_report		Not Loaded			Loaded
old_api					Loaded

Loading Packages

To load a Quartus II Tcl package, use the following Tcl command:

```
load_package [-version <version number>] <package name>.
```

This command is similar to the `package require` Tcl command (described in [Table 3–7 on page 3–31](#)), but you can easily alternate between different versions of a Quartus II Tcl package with the `load_package` command.



For additional information on these and other Quartus II command-line executables, see 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. They are listed in [Table 3–3](#). Each executable supports different sets of Tcl packages. See the following table to determine the appropriate executable to run your script.

Table 3–3. 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 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	Use the Quartus II Simulator to simulate designs with Tcl testbenches

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, see the *Command-Line Scripting* chapter in Volume 2 of the *Quartus II Handbook*.

Command-Line Options: -t, -s, and --tcl_eval

[Table 3–4](#) lists three command-line options you can use with executables that support Tcl.

Table 3–4. 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: <code>quartus_sh --tcl_eval help -pkg project</code>

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>/bin/tcl_packages/tcllib-1.4/cmdline` directory.



The Quartus II software beginning with version 4.1 supports the `argv` variable. In previous software versions, script arguments are accessed in the `quartus(args)` global variable.

Interactive Shell Mode

Running an executable with the `-s` option starts an interactive Tcl shell session that displays a `tcl>` prompt. Everything you type in the Tcl shell is immediately interpreted by the shell. You can run a Tcl script within the interactive shell with the following command:

`source <script name> [<script arguments>]` ↵

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.

Using the Quartus II Tcl Console Window

You can run Tcl commands directly in the Quartus II Tcl Console window. To open the window, choose **Utility Windows > Tcl Console** (View menu). By default, the Tcl Console 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 appear in the Quartus II Tcl Console window.

Examples

Most chapters in the *Quartus II Handbook* include information about scripting support. They include feature-specific examples and script information. For scripting help on a specific feature, see the corresponding chapter in the handbook.

If you are an advanced Tcl scripting user, you can refer to some Tcl scripts included with the Quartus II software and modify them to suit your needs. The Design Space Explorer (DSE), Quartus II Command-Line and Tcl API reference, and QFlow are written with Tcl and Tk. Files for those scripts are located in the `<Quartus II installation>/bin/tcl_scripts` directory.

Natural Bus Naming

The Quartus II software, beginning with version 4.2 supports natural bus naming. Natural bus naming means that square brackets used to specify bus indexes in hardware description languages do not need 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 this example:

```
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.

Accessing Command-Line Arguments

Virtually all Tcl scripts must 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. Here is a code example that prints all the arguments in the `quartus(args)` variable:

```

set i 0
foreach arg $quartus(args) {
    puts "The value at index $i is $arg"
    incr i
}

```

If you save these commands in a Tcl script file called **print_args.tcl**, you see the following output when you type this command at a command prompt:

```
quartus_sh -t print_args.tcl my_project 100MHz
```

```
The value at index 0 is my_project
```

```
The value at index 1 is 100MHz
```

The Quartus II software beginning with version 4.1 supports the Tcl variables `argv`, `argc`, and `argv0` for command-line argument access. [Table 3–5](#) shows equivalent information for earlier versions of the software.

Table 3–5. Quartus II Support for Tcl Variables

Beginning With Version 4.1	Equivalent Support in Previous Software Versions
<code>argc</code>	<code>llength \$quartus(args)</code>
<code>argv</code>	<code>quartus(args)</code>
<code>argv0</code>	<code>info nameofexecutable</code>

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>`.

The following example uses the `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 will see the following output when you type this command at a command prompt:

```
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. The following example 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.

```
package require cmdline
variable ::argv0 $::quartus(args)
set options {
    { "project.arg" "" "Project Name" } \
    { "revision.arg" "" "Revision Name" } \
}
array set optshash [::cmdline::getoptions ::argv0 $options]

if {[project_exists $optshash(project)]} {
    if {[string equal "" $optshash(revision)]} {
        project_open $optshash(project) -current_revision
    } else {
        project_open $optshash(project) -revision $optshash(revision)
    }
} else {
    puts "Project $optshash(project) does not exist"
    exit 1
}
# The rest of your script goes here
```



For more information on the **cmdline** package, refer to the documentation for the package at <Quartus II installation directory>/bin/tcl_packages/tcllib-1.4/doc/cmdline.html

Creating Projects & 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. Choose **Generate Tcl File for Project** (Project menu) 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.

The following example shows how to create a project, make assignments, and compile the project. It uses the **fir_filter** tutorial design files.

```
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.

Compiling Designs

You can run the Quartus II command-line executables from Tcl scripts either with the included `::quartus::flow` package to run various Quartus II compilation flows, or by running each executable directly.

The `::quartus::flow` Package

The `::quartus::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 `::quartus::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 Tcl's `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 `::quartus::flow` package to ensure assignments are exported to the project database and settings file.



Altera recommends using 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> ↵
```

Extracting Report Data

Once a compilation finishes, you may need 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.

You can use commands that access report data one row at a time, or a 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. At times you may need to 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`.

The following example 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**.

```
load_report
set report_panel_name "Timing Analyzer||Timing Analyzer Summary"
set num_rows [get_number_of_rows -name $report_panel_name]
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"]
```

```

for {set i 1} {$i < $num_rows} {incr i} {
    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

```

Using Collection Commands

Some Quartus II Tcl functions return very large sets of data which 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 help 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. The following example prints all instance assignments in an open project.

```

set all_instance_assignments [get_all_instance_assignments -name *]
foreach_in_collection asgn $all_instance_assignments {
    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. The following example prints the quantity of global assignments in an open project:

```

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"

```

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 following script example.

```
# 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
```

EDA Tool Assignments

You can target EDA tools for a project in the Quartus II software in Tcl by using 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 (<code>EDA_DESIGN_ENTRY_SYNTHESIS_TOOL</code>)	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 (<code>EDA_SIMULATION_TOOL</code>)	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

Table 3–6. EDA Interface Options in the Quartus II Software (Part 2 of 2)

Option	Acceptable Values
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
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 NC-Sim Verilog simulation output, EDA_SIMULATION_TOOL should be set to target NC-Sim Verilog as the desired output, as shown below:

```
set_global_assignment -name eda_simulation_tool\
"NcSim (Verilog HDL output from Quartus II)"
```

The following example shows compilation of the fir_filter design files, generating a VHDL Output File (.vho) output for NC-Sim Verilog simulation:

```
# 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, see “EDA Tool Setting Section (Settings and Configuration Files)” in 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.

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.

The following script creates a project with the **lockmult** directory tutorial files, 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.

```
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 }
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 this example:

```
proc assign_virtual_pins { skips } {
    # Analysis and elaboration must be run first to get the pin names
    execute_flow -analysis_and_elaboration
    set name_ids [get_names -filter * -node_type pin]
    foreach_in_collection name_id $name_ids {
        set hname [get_name_info -info full_path $name_id]
        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 (.qsf) 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.

The following script performs those steps:

```
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 on the LogicLock design methodology, see the *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*.

Using the post_message Command

To print messages that are formatted like Quartus II software messages, use the `post_message` command. 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 using the Quartus II software in Windows, you can add color to messages printed with the `post_message` command. Add the following line to your `quartus2.ini` file:

```
DISPLAY_COMMAND_LINE_MESSAGES_IN_COLOR = on
```

This example shows how to use the `post_message` command:

```
post_message -type warning "Design has gated clocks"
```

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 below:

```
C:\>quartus_sh -s
Info: ****
Info: Running Quartus II Shell
Info: Version 4.0 Internal Build 131 10/06/2003 SJ Full Version
Info: Copyright (C) 1991-2003 Altera Corporation. All rights reserved.
Info: Quartus is a registered trademark of Altera Corporation in the
Info: US and other countries. Portions of the Quartus II software
Info: code, and other portions of the code included in this download
Info: or on this CD, are licensed to Altera Corporation and are the
Info: copyrighted property of third parties who may include, without
Info: limitation, Sun Microsystems, The Regents of the University of
Info: California, Softel vdm., and Verific Design Automation Inc.
Info: Warning: This computer program is protected by copyright law
Info: and international treaties. Unauthorized reproduction or
Info: distribution of this program, or any portion of it, may result
Info: in severe civil and criminal penalties, and will be prosecuted
Info: to the maximum extent possible under the law.
Info: Processing started: Thu Nov 20 19:54:12 2003
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 -pkg <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>
```

At the Tcl prompt, create a new project called `fir_filter` with a revision name called `filtref` by typing the following command:

```
tcl> 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.

Since 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:

```
tcl> set_global_assignment -name family Cyclone
```



To learn more about assignment names that you can use with the `-name` option, see “Settings and Configuration Files Introduction” in 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:

```
tcl> load_package flow
```

```
1.0
```

For additional help on the `::quartus::flow` package, view the command-line help at the Tcl prompt by typing:

```
tcl> help -pkg ::quartus::flow
```

This sample shows an alternative command and the resulting 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 on the `:quartus::flow` package and the commands that are available with the package. To read help on the `execute_flow` Tcl command, short help displays the options:

```
tcl> execute_flow -h ↵
```

Long help displays additional information and example usage:

```
tcl> execute_flow -long_help ↵
```

or

```
tcl> 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 the following example:

```
tcl> execute_flow -compile ↵
Info:*****
Info: Running Quartus II Analysis & Synthesis
Info: Version 4.0 SJ Full Version
Info: Processing started: Mon Nov 18 09:30:47 2003
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 choosing **Start Compilation** (Processing menu) in the Quartus II GUI.

When you are finished with a project, close it using the `project_close` command:

```
tcl> project_close ↵
tcl>
```

Then, to exit the interactive Tcl shell, type `exit`.

```
tcl> exit ↵
```

Getting Help on Tcl & Quartus II Tcl APIs

Quartus II Tcl help allows easy access to information on the Quartus II Tcl commands. To access the help information, type `help` at a command prompt, as shown below (with sample output):

```
tcl> help
-----
-----
Available Quartus II Tcl Packages:
-----
Loaded           Not Loaded
-----           -----
::quartus::device   ::quartus::flow
::quartus::misc    ::quartus::report
::quartus::project

* Type "help -tcl"
  to get an overview on Quartus II Tcl usages.
-----
tcl>
```

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.

Table 3–7 summarizes the help options available in the Tcl environment.

Table 3–7. Help Options Available in the Quartus II Tcl Environment (Part 1 of 2)	
Help Command	Description
<code>help</code>	To view a list of available Quartus II Tcl packages, loaded and not loaded.
<code>help -tcl</code>	To view a list of commands used to load Tcl packages and access command-line help.
<code>help -pkg <package_name> [-version <version number>]</code>	<p>To view help for a specified Quartus II package that includes the list of available Tcl commands. For convenience, you can omit the <code>::quartus::</code> package prefix, and type <code>help -pkg <package name> ↵</code>. If you do not specify the <code>-version</code> 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.</p> <p>Examples:</p> <pre>help -pkg ::quartus::p ↵ help -pkg ::quartus::project ↵ help -pkg project rhelp -pkg project -version 1.0 ↵</pre>

Table 3–7. Help Options Available in the Quartus II Tcl Environment (Part 2 of 2)

Help Command	Description
<command_name> -h or <command_name> -help	<p>To view short help for a Quartus II Tcl command for which the package is loaded.</p> <p>Examples:</p> <pre>project_open -h ↵ project_open -help ↵</pre>
package require ::quartus::<package name> [<version>]	<p>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.</p> <p>Example:</p> <pre>package require ::quartus::project 1.0 ↵</pre> <p>This command is similar to the <code>load_package</code> command. The advantage of using <code>load_package</code> is that you can alternate freely between different versions of the same package. Type <code><package name> [-version <version number>]</code> ↵ to load a Quartus II Tcl package with the specified version. If the <code>-version</code> option is not specified, the latest version of the package is loaded by default.</p> <p>Example:</p> <pre>load_package ::quartus::project -version 1.0 ↵</pre>
help -cmd <command name> [-version <version number>] or <command name> -long_help	<p>To view long help for a Quartus II Tcl command. Only “<command name> -long_help” requires that the associated Tcl package is loaded.</p> <p>If you do not specify the <code>-version</code> option, help for the currently loaded package is displayed by default.</p> <p>If the package for which you want help is not loaded, help for the latest version of the package is displayed by default.</p> <p>Examples:</p> <pre>project_open -long_help ↵ help -cmd project_open ↵ help -cmd project_open -version 1.0 ↵</pre>
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. See “ The Tcl/Tk GUI Help Interface ” for more information.

There are two types of help for Tcl commands:

- For information on the usage and a brief description of a Tcl command type, use the `-help` option. (The `-h` command-line option may be used instead of `-help`, if preferred.) If the Tcl command is part of a Tcl package that is not loaded, using the `-help` option returns “invalid command name” as an error message.
- For more detailed help on a given Tcl command, use the `-long_help` option or type `help -cmd < Tcl command name >`. If the Tcl command is part of a Tcl package that is not loaded, typing `<command name> -long_help` returns the error message “invalid command name”.



Using the `-cmd` option does not require that the specific Tcl command be loaded. Only the `-long_help` option requires that the relevant Tcl package be loaded.

The Tcl/Tk GUI Help Interface

For a complete list of package and commands available with the Quartus II software, open the help browser that lists all Quartus II command-line executables and Tcl API packages and their respective commands. To open the help browser, type the following command at a command prompt:

```
C:\> quartus_sh --qhelp
```

This runs a Tcl/Tk script that provides help for Quartus II command-line executables and Tcl API packages and commands.



For more information on this utility, see the *Command-Line Scripting* chapter in Volume 2 of the *Quartus II Handbook*.

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 by using the Quartus II Tcl console or by using the `quartus_cmd` executable at the 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 a third-party EDA tool that does not generate Tcl scripts for the most recent version of the Quartus II software.



Altera recommends creating all new projects and Tcl scripts with the latest version of the Quartus II Tcl API.

References

For more information on using Tcl, see 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 www.altera.com/support/examples/tcl/tcl.html
- Tcl Developer Xchange at tcl.activestate.com

qii52012-1.2

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 are required to keep track of their changes to the project. To help designers manage their FPGA designs, the Quartus® II software provides the Revisions, Copy Project, and Version-Compatible Database features.

In the Quartus II software, a revision is one set of assignments and settings. A project can have multiple revisions, each with their own set of assignments and settings. Creating multiple revisions allows you to change assignments and settings while preserving 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 databases to be compatible across different versions of the Quartus II software, thus avoiding unnecessary compilations. This chapter also discusses how to smoothly migrate your projects from one computing platform to another.

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 ability allows you to explore different assignments and settings for your design and then compare the results. There are several ways to use the revisions feature.

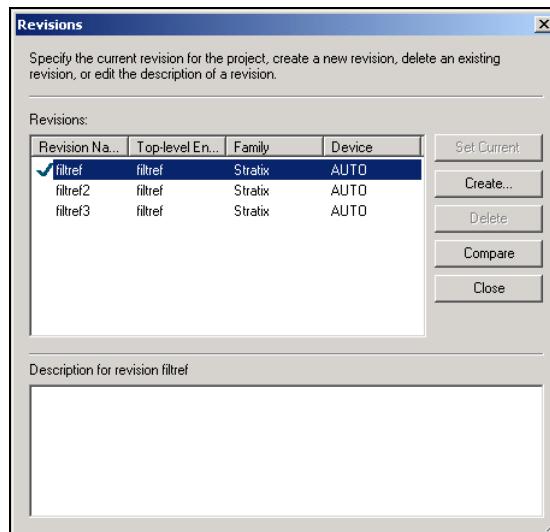
- The first method is to create a new revision of your design that is not based on any previous revision. For example, early in your design you may want to create a revision containing assignments that target area optimization and another revision containing assignments that target f_{MAX} optimization.

- The second method is to create a new revision based on an existing revision and then try new settings and assignments in the new revision. Your new revision will already include all the assignments and settings made in the previous revision. If you are not satisfied with the results from the new revision, you can revert to the original revision.
- The third method is to compare different compilation results from different revisions, select the revision that best meets your design requirements, create a new revision based on the best revision (that you just selected), and perform further optimizations on the new revision until the design meets all design requirements.

Creating & Deleting Revisions

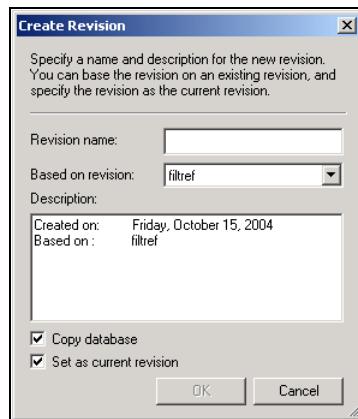
All Quartus II assignments and settings are stored in the Quartus Settings File (.qsf). Each time you create a new revision of a project, the Quartus II software creates a new QSF and adds the name of the new revision to the list of revisions in the Quartus Project File (.qpf). You manage revisions with the **Revisions** dialog box (Figure 4–1), which allows you to set the current revision, create, delete, and compare revisions in a project.

Figure 4–1. Revisions Dialog Box



To Create a Revision

1. If you have not already done so, create a new project or open an existing project by choosing **New Project Wizard** or **Open Project** (File menu).
2. Choose **Revisions** (Project menu).
3. If you want 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. Create Revision Dialog box

5. In the **Create Revision** dialog box shown in [Figure 4–2](#), type the name of the new revision in the **Revision name** box.
6. If you want to base the new revision on an existing revision for the current design and you did not select the correct revision in Step 3 (from [Figure 4–1](#)), select the revision in the **Based on revision** list as shown in [Figure 4–2](#).

or

If you do not want to base the new revision on an existing revision for the current design, select the “blank entry” in the **Based on revision** list.

7. Optionally, edit the description of the revision in the **Description** box.

8. If you based the new revision on an existing revision for the current design, and you want the new revision to contain the database information from the existing revision, turn on **Copy database**.
9. If you want to specify the new revision as the current revision, turn on **Set as current revision**. Click **OK**.
10. In the **Revisions** dialog box (see [Figure 4–1](#)), click **Close**.

To Delete a Revision That is a Design's Current Revision

[Figure 4–1](#) shows **filtref** as the current revision. As the figure shows, the **Delete** button is not available which prevents you from deleting the current revision file. To delete the current revision, you must temporarily set a different revision to be the current revision. The following example, shows how to delete **filtref**.

1. If you have not already done so, open an existing project by selecting **Open Project** (File menu).
2. Choose **Revisions** (Project menu).
3. In the **Revisions** list (see [Figure 4–1](#)), **filtref** is the current revision. Temporarily select a different revision and make it the current revision.
4. Choose **Set Current**.
5. Select **filtref** and Click **Delete**.

To Delete a Revision That is not a Design's Current Revision

In this example, you delete **filtref2** ([Figure 4–1](#)).

1. If you have not already done so, open an existing project by choosing **Open Project** (File menu).
2. Choose **Revisions** (Project menu).
3. In the **Revisions** list, select the revision you want to delete and click **Delete**. In this example, it is **filtref2**.

Comparing Revisions

You can compare the results of multiple revisions side by side with the **Compare Revisions** dialog box. To compare all revisions in a single window, click **Compare** in the **Revisions** dialog box (Project menu). In the **Compare Revisions** dialog box (see [Figure 4-3](#)), the results of each revision in three categories (Analysis & Synthesis, Fitter, and Timing Analyzer) are compared side by side.

Figure 4-3. Compare Revisions Dialog Box

The screenshot shows the 'Compare Revisions' dialog box with two tabs: 'Results' and 'Assignments'. The 'Results' tab is selected, displaying a grid of comparison results for three revisions: 'Revision filref_new' (highlighted in yellow), 'Revision filref_opt' (highlighted in green), and 'Revision filref' (highlighted in blue). The grid is organized into three main sections: Analysis & Synthesis, Fitter, and Timing Analyzer. Each section contains several parameters with their values and status indicators.

	C:/tools/quartus/... Revision filref_new	C:/tools/quartus/... Revision filref_opt	C:/tools/quartus/... Revision filref
Analysis & Synthesis			
Flow Status	Successful - Tue...	Successful - Tue...	Successful - Tue...
Quartus II Version	4.2 Internal Build ...	4.2 Internal Build ...	4.2 Internal Build ...
Revision Name	filref_new	filref_opt	filref
Top-level Entity Name	filref	filref	filref
Family	Cyclone	Cyclone	Cyclone
Meet timing requirements	N/A	N/A	N/A
Device	EP1C12F256C6	EP1C6F256C6	EP1C6F256C6
Timing Models	Final	Final	Final
Total logic elements	106 / 12,060 (< ...)	106 / 5,980 (1 %)	102 / 5,980 (1 %)
Total pins	22 / 185 (11 %)	22 / 185 (11 %)	22 / 185 (11 %)
Total virtual pins	0	0	0
Total memory bits	0 / 239,616 (0 %)	0 / 92,160 (0 %)	0 / 92,160 (0 %)
Total PLLs	0 / 2 (0 %)	0 / 2 (0 %)	0 / 2 (0 %)
Timing Analyzer			

In addition to viewing the results of each revision, you can also show the assignments of each revision. Click the **Assignments** tab of the **Compare Revisions** dialog box to view all assignments applied to each revision (see [Figure 4-3](#)). You can gain a 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).

Creating a new version of your project with the Quartus II software involves creating a copy of your project and then editing your design files. For example, you have a design that is compatible with a 32-bit data bus and now you need to create a new version of the design to interface

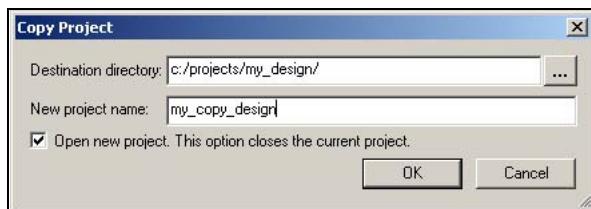
with a 64-bit data bus. To solve this problem, create a new version of your project by choosing **Copy Project** (Project menu), and making the necessary design changes.

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 involves creating a copy of your project and then replacing the previous netlist file with the newly generated netlist file. Use the **Copy Project** command (Project menu) to create a copy of your design and use the **Add/Remove Files in Project** command (Project menu) to add and remove design files, if necessary.

To create a new version of your project, use the **Copy Project** command (Project menu).

1. Choose **Copy Project** (Project menu). This opens the **Copy Project** dialog box (see [Figure 4-4](#)).
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 the **Open new project. This option closes the current project** option.
5. Click **OK**.

Figure 4-4. Copy Project Dialog Box



Archiving Projects With the Quartus II Archive Project Feature

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. You also have the option to include additional files and the project database. The QAR file contains all the files required to perform a full compilation to restore the original results.

A single project can contain hundreds of files, and it may be difficult to transfer a project between engineers. With the archive file generated by the Archive Project feature (see [Figure 4–5](#)), you can easily share projects between engineers.

Figure 4–5. Archive Project Dialog Box



Archive a Project

To archive a project, perform the following steps:

1. If you have not already done so, create a new project or open an existing project by selecting **New Project Wizard** or **Open Project** (File menu).
2. Altera® recommends that you perform analysis and elaboration before archiving a project to ensure that all design files are located and archived.
3. Choose **Archive Project** (Project menu).
4. In the **Archive file name** box, type the file name of the Quartus II Archive File (.qar) you wish to archive, or select a QAR name with **Browse (...)**.

5. Turn on **Archive current active revision only** to archive only the currently active revision. If you do not turn on this option, all revisions within the project are included in the project archive.
6. Select one of the following items from the **Include the following optional database files** section of the Archive Project dialog box (Figure 4–5).
 - 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 edit the contents of the QAR file.
9. Click **OK**.

Restore an Archived Project

To restore an archived project, complete the following steps:

1. Choose **Restore Archived Project** (Project menu).
2. In the **Archive file name** box, type the file name of the QAR file you wish to restore, or select a QAR file with **Browse (...)**.
3. In the **Destination folder** box, specify the path to the directory into which you wish to restore the contents of the QAR file, or select a directory with **Browse (...)**.
4. Click **Show log** to view the Quartus II Archive Log File (.qarlog) for the project you are restoring from the QAR file.
5. Click **OK**.

6. If you did not include the compilation and simulation database files in the project archive, [Figure 4–5 on page 4–7](#), 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 4.2 without having to recompile your design.

Perform the following steps to export a version-compatible database:

1. Choose **Export Database** (Project menu).
2. Specify a path in the **Export Directory** box.
3. Click **OK**.

To import a version-compatible database, perform the following steps:

1. Choose **Import Database** (Project menu).
2. Browse to the directory to which the database was previously exported. The default directory is *<project name>\export_db*.
3. Click **OK**.

To save the database in a version-compatible format during every compilation, perform the following steps:

1. Choose **Settings** (Assignments menu).
2. Select the **Compilation Process** page.
3. Turn on the **Export version-compatible database** option.
4. Browse to the directory in which you want to save the database.
5. Click **OK**.

Quartus II Project Platform Migration

When moving your project from one computing platform to another, there are several things that should be considered. This section describes the following considerations:

- Filenames and hierarchy
- Quartus II search path precedence rules
- Specifying libraries
- Quartus II-generated files for 3rd-party EDA tools
- Migrating database files

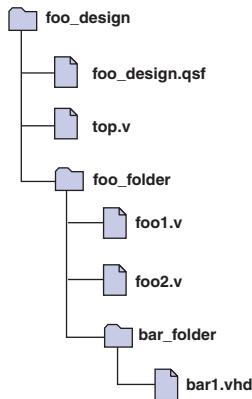
Filenames and Hierarchy

To ensure smooth migration across platforms, you need to 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 systems file systems are case-sensitive. When writing scripts, make sure to specify paths exactly, even if the current operating system is not case-sensitive. For best results, always use lowercase letters when naming files.
- Use a character set common to all the platforms that you may use.
- It's not necessary to convert between forward-slash "/" and back-slash "\\" path separators in the QSF, 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 may use.

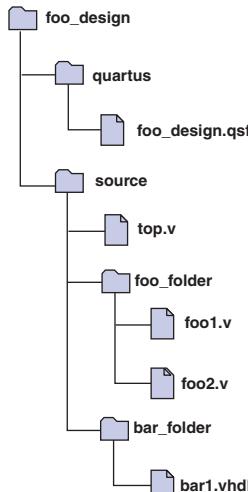
You can specify files and directories inside a Quartus II project as paths relative to the project directory. For instance, for a project titled **foo_design** with a directory structure shown in [Figure 4-6](#), specify the source files as: **top.v**, **foo_folder/foo1.v**, **foo_folder/foo2.v**, and **foo_folder/bar_folder/bar1.vhdl**.

Figure 4–6. All-Inclusive Project Directory Structure



If the directory structure is such that the QSF is in a directory that is separate from the source files there are two options. If the source files are very near the Quartus II project directory, it may be convenient to express relative paths using ".." notation. For example given the directory structure shown in [Figure 4–7](#), you can specify `top.v` as `./source/top.v` and `foo1.v` as `../source/foo/foo_folder/foo1.v`.

Figure 4–7. Quartus II Project Directory Separate from Design Files





When you copy a directory structure to a different platform, make sure that any subdirectories occur in the same hierarchy as on the original platform.

Specifying Libraries

You can also specify the directory (or directories) containing the source as a library that Quartus II searches when you compile your project. A library is a directory containing design files used by your Quartus II project. There are two kinds of libraries that can be specified in the Quartus II software: user libraries, which apply to a specific project, and global libraries, which are used for all projects. Use the procedures in this section to specify user or global libraries.

All files in the directories specified as libraries are relative to them. For instance, if you specify a path `/user_lib1/foo1.v`, if the `user_lib1` directory is specified as a user library in the project, the `foo1.v` file can be specified in the QSF as simply `foo1.v`. The Quartus II software searches the directories that are specified as libraries and finds the file.

How to Specify User Libraries

Specify user libraries in the GUI on the **User Libraries** page of the **Settings** dialog box (Assignments menu). Type the name of the directory in the **Library name** box.

How to Specify Global Libraries

Specify global libraries in the GUI on the **User Libraries** page of the **Options** dialog box (Tools menu). Type the name of the directory in the **Library name** box.



Whenever you specify a directory name in the GUI or in Tcl, the name you use will be maintained verbatim in the QSF rather than resolved to an absolute path.

When using the **Browse** button in either the **Settings** dialog box or the **Options** dialog box to select a directory, if the directory is outside of the project directory, the path returned in the dialog box is an absolute path. You can change absolute path to relative path before adding it to the list.

When copying project directories that are specified as user libraries, you must either copy your user library files along with the project directory or make sure 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 in the following order:

1. The project directory (The directory in which the QSF is found.)
2. The project's **db** directory.
3. User libraries, in the order specified in the USER_LIBRARIES setting of the QSF for the current revision.
4. Global libraries, in the order specified in the the USER_LIBRARIES setting of the QSF for the current revision.
5. The Quartus II software **libraries** directory.

Quartus II-Generated Files for 3rd 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 may cause problems for migration.

Scripting Support

You can run the procedures and make the settings described in this chapter in a Tcl script. 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 **quartus_sh --qhelp** at a system command prompt to run the Quartus II Command-Line and Tcl API Help browser. For more information on Quartus II scripting support, including examples, see the *Tcl Scripting* and *Command-Line Scripting* chapters 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, see “[Creating & Deleting Revisions](#)” on page 4-2.

Creating Revisions

The following Tcl command creates a new revision called `speed_ch` based on a revision called `chiptrip` and sets it 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 set 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. For more information about archiving projects, see “[Archiving Projects With the Quartus II Archive Project Feature](#)” on page 4-6.

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 to <project name>:

```
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, see “[Restore an Archived Project](#)” on page 4–8.

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 & 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, see “[Version-Compatible Databases](#)” on page 4–9.



The **flow** package and the **database_manager** package contain commands to manage version-compatible databases.

Use the following commands from the **database_manager** package to import or export version-compatible databases.

```
export_database <directory>
import_database <directory>
```

Use the following commands available in 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

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 for 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 paths of user 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 sets of different assignments and settings. The Copy Project feature allows you to create a new version of your design by copying a set of design files and one or more revisions.

The Quartus II Version-Compatible Database feature saves compilation time when moving to updated versions of the Quartus II software. These features in the Quartus II software help facilitate efficient management of your design to accommodate today's more sophisticated FPGA designs.

This section describes the design flow to assign and analyze pin-outs using the **Start I/O Assignment Analysis** command in the Quartus® II software, both with and without a complete design.

This section includes the following chapter:

- [Chapter 5, I/O Assignment Planning & Analysis](#)

Revision History

The table below shows the revision history for [Chapter 5](#).

Chapter(s)	Date / Version	Changes Made
5	Jan. 2005 v2.2	<ul style="list-style-type: none">● Updated to include information on migrating projects between different computing platforms.● Added I/O Rules Checked by I/O Assignment Analysis section including related rules table
	Dec. 2004 v2.1	Reorganized chapter and updated information for 4.2.
	June 2004 v2.0	<ul style="list-style-type: none">● Scripting support section added.● Updated coding examples.
	Feb. 2004 v1.0	Initial release.

qii52004 - 2.2

Introduction

Today's FPGAs support multiple I/O standards and have high pin counts. You must be able to make pin assignments efficiently for designs in these advanced devices. You also need the ability to easily check the legality of the pin assignments to ensure that the pin-out does not violate any board layout guidelines such as pin spacing and current consumption limitations.

This chapter describes a 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.

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.

You can use the **Start I/O Assignment Analysis** command on designs targeting Stratix® II, Stratix, Stratix GX, MAX® II, Cyclone™ II, and Cyclone device families.

I/O Assignment Planning & Analysis Design Flows

The I/O assignment planning and analysis design flows depend on whether your project contains design files.

- When the board layout must be complete before starting the FPGA design, use the flow shown in [Figure 5–1 on page 5–3](#). 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 5–3 on page 5–5](#). This flow thoroughly checks the legality of your pin assignments against any design files provided. For more information on creating assignments, see the *Assignment Editor* chapter in Volume 2 of the *Quartus II Handbook*.

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 repeatedly since it completes in a short time.

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 I/O standard assigned, 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 phase-locked loops (PLLs), low-voltage differential signals (LVDS), or gigabit transceiver blocks.

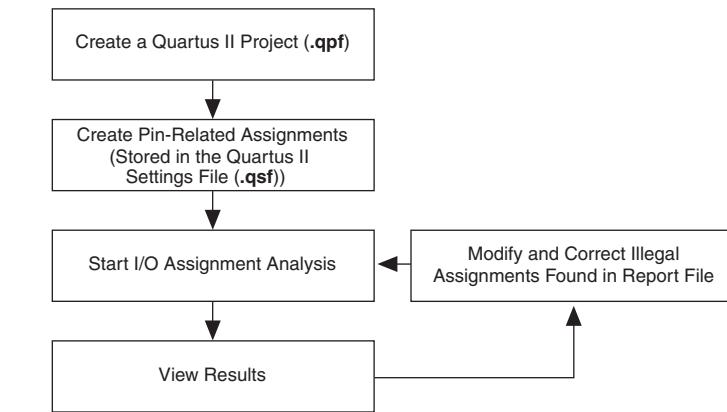
Design Flow Without Design Files

During the early stages of development of an FPGA device, board layout engineers may request preliminary or final pin-outs. It is time consuming to manually check to see whether the pin-outs violate any design rules. Instead, you can 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 did not violate design rules.

The I/O assignment analysis command is able to perform limited checks against 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 to each pin using the Assignment Editor. After you assign an I/O standard to each reserved pin, you run the I/O assignment analysis to ensure that there are no I/O standard conflicts in each I/O bank.

Figure 5–1. Assigning & Analyzing Pin-Outs Without Design Files

You can assign and analyze pin-outs using the **Start I/O Assignment Analysis** command without design files by following these steps:

1. Create a Quartus II project.
2. Use the **Assignment Editor** or Tcl commands to create pin locations and related assignments. For the I/O assignment analysis to determine the type of a pin, you must reserve your I/O pins. See “Reserving Pins” on page 5–8.
3. Choose **Start > Start I/O Assignment Analysis** (Processing menu) to start the analysis.



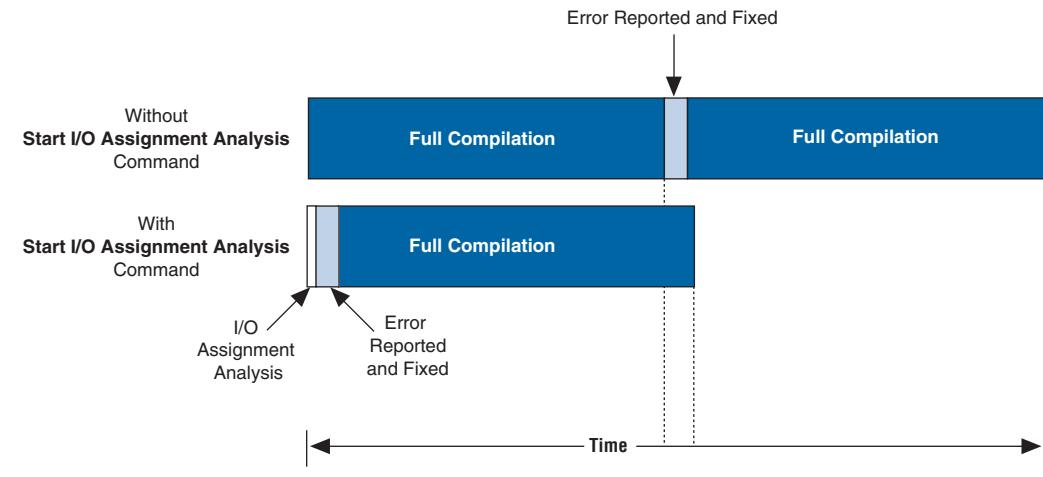
For information on using a Tcl script or command prompt to start the analysis, see the section “Scripting Support” on page 5–12.

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.
6. Rerun the **Start I/O Assignment Analysis** command until all errors are corrected.

Design Flow 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, you can run the **Start I/O Assignment Analysis** command after performing analysis and synthesis and before performing a full compilation. Typically, the analysis takes a short time. Figure 5–2 shows the benefits of using the **Start I/O Assignment Analysis** command.

Figure 5–2. Saving Compilation Time With the Start I/O Assignment Analysis Command

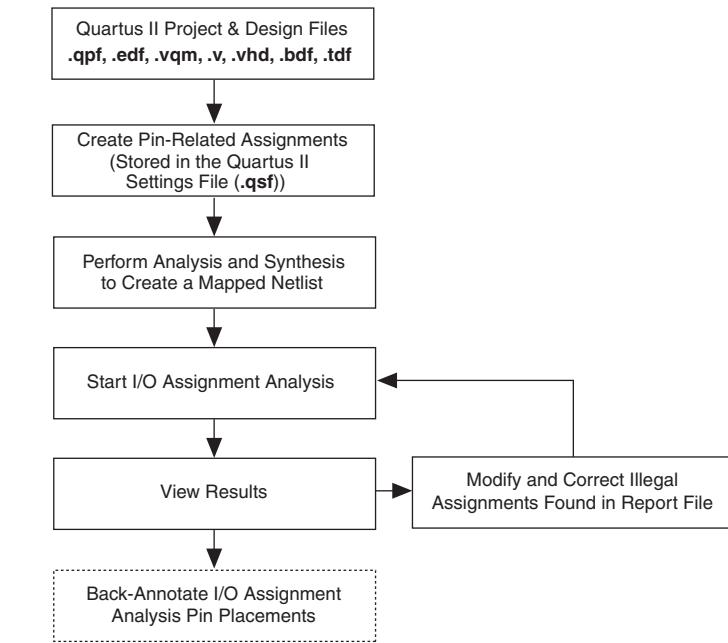


The rules that are checked by the I/O assignment analysis depend on the completeness of the design. With a complete design, the **Start I/O Assignment Analysis** command thoroughly checks the legality of all pin-related assignments. With a partial design, which can just be the top-level wrapper file, 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 you design the clock to drive a PLL that has not yet been instantiated in the design. Because the **Start I/O Assignment Analysis** command is unaware of 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.

Analyze as much of the design as possible, especially logic that connects to pins, to obtain better coverage. 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 5–3. Assigning & 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 Quartus II project and include your design files in the project.
2. Create pin-related assignments with the **Assignment Editor**.



You can also create pin-related assignments by importing them from a Comma-Separated Value file (.csv), executing Tcl commands, editing the Quartus II Settings File (.qsf) directly, or by dragging and dropping pins to a location in the timing closure floorplan.

3. Choose **Start > Start Analysis & Synthesis** (Processing menu) to generate an internal mapped netlist.



For information on using a Tcl script or command prompt to start the analysis, see the section “[Scripting Support](#)” on page 5–12.

4. Choose **Start > Start I/O Assignment Analysis** (Processing menu) to start the analysis.
5. View the messages in the Compilation Report or in the Messages window.
6. Use the **Assignment Editor** to correct any errors and violations reported.
7. Use the **Start I/O Assignment Analysis** command until all errors are corrected.

I/O Rules Checked by the I/O Assignment Analysis

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 as many design files as possible and all pin-related assignments in your Quartus II project.

[Tables 5–1](#) and [5–2](#) list a subset of the I/O rule checks performed when you execute an I/O Assignment Analysis with and without design files. For more detailed information on each I/O rule, please refer to the “Selectable I/O Standards” chapter in the respective device handbook.

Table 5–1. General I/O Related Rules (Part 1 of 2)

Rule	Description	Device(1) 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 V _{CCIO} is required from the pins assigned to the I/O bank.	All	No
I/O bank V _{REF} voltage compatibility	Checks that no more than one V _{REF} is required from the pins assigned to the I/O bank.	All	No
I/O standard and location conflicts	Checks if the pin location supports the assigned I/O standard.	All	No

Table 5–1. General I/O Related Rules (Continued) (Part 2 of 2)

Rule	Description	Device(1) Families	HDL Required?
I/O standard and signal direction conflicts	Checks if the pin location supports the I/O standard assigned and the 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 to see if the drive strength assignments is within the specifications of the I/O standard.	All	No
Drive strength and location conflicts	Checks to see if the pin location supports the assigned drive strength.	All	No
BUSHOLD and location conflicts	Checks if the pin location supports BUSHOLD (e.g., dedicated clock pins do not support BUSHOLD).	All	No
WEAK_PULLUP and location conflicts	Checks if the pin location supports WEAK_PULLUP (for example, dedicated clock pins do not support WEAK_PULLUP)	All	No
Electromigration check	Checks if the combined drive strength of consecutive pads does not exceed 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 if 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 if pins connected to PLL are assigned to the dedicated PLL pin locations.	All	Yes

Notes to Table:

- (1) “All” includes the following device families: Cyclone II, Cyclone, Stratix II, Stratix GX, Stratix, MAX II, and Hardcopy devices.

Table 5–2. SSN Related Rules			
Rule	Description	Device(1) Families	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 GX, Stratix II	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 if 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 if single-ended output pins are a certain distance away from a VREF pad.	Cyclone & Cyclone II	No
Single-ended input is required to be a certain distance away from a differential I/O pin	Checks if single-ended output pins are a certain distance away from a differential I/O pin.	Cyclone & Cyclone II	No
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 if too many outputs are in a VREFGROUP.	All	No

Notes to Table:

- (1) “All” includes the following device families: Cyclone II, Cyclone, Stratix II, Stratix GX, Stratix, MAX II, and Hardcopy devices.

Inputs for I/O Assignment Analysis

The **Start I/O Assignment Analysis** command reads an internal mapped netlist and a Quartus II Settings File (.qsf). All assignments are stored in the single QSF.

If you do not have any design files, the **Start I/O Assignment Analysis** command reads all the assignments in the QSF.

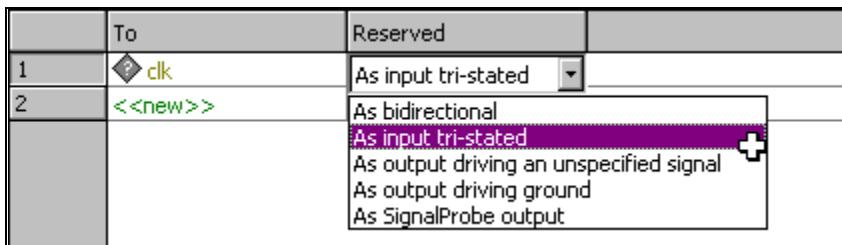
If you have a partial or complete design, the **Start I/O Assignment Analysis** command reads all the assignments in the QSF and the mapped netlist file.

Reserving Pins

If you do not have any design files, you can reserve a pin location in addition to other pin-related assignments for that location. Reserving pins is necessary so that the **Start I/O Assignment Analysis** command understands the pin type (input, output, or bidirectional) and correctly

analyzes the pins. You can reserve a pin by choosing **Assignment Editor** (Assignments menu) and selecting **Reserved Pin** from the Category list. In the spreadsheet, type in the pin name and select from the reserved list (see [Figure 5–4](#)).

Figure 5–4. Reserving an Input Pin With the Assignment Editor



The screenshot shows a portion of the Assignment Editor interface. A table has two rows. Row 1 contains a pin number '1' and a 'To' column with a symbol for a clock input and the text 'clk'. The 'Reserved' column dropdown is open, showing 'As input tri-stated' as the selected option. Row 2 contains a pin number '2' and a 'To' column with the text '<<new>>'. The 'Reserved' column dropdown is also open, displaying several options: 'As bidirectional', 'As input tri-stated' (which is highlighted with a purple background), 'As output driving an unspecified signal', 'As output driving ground', and 'As SignalProbe output'. A small blue plus sign icon is located to the right of the dropdown menu.



For more information on using the **Assignment Editor**, see the *Assignment Editor* chapter in Volume 2 of the *Quartus II Handbook*.

Location Assignments

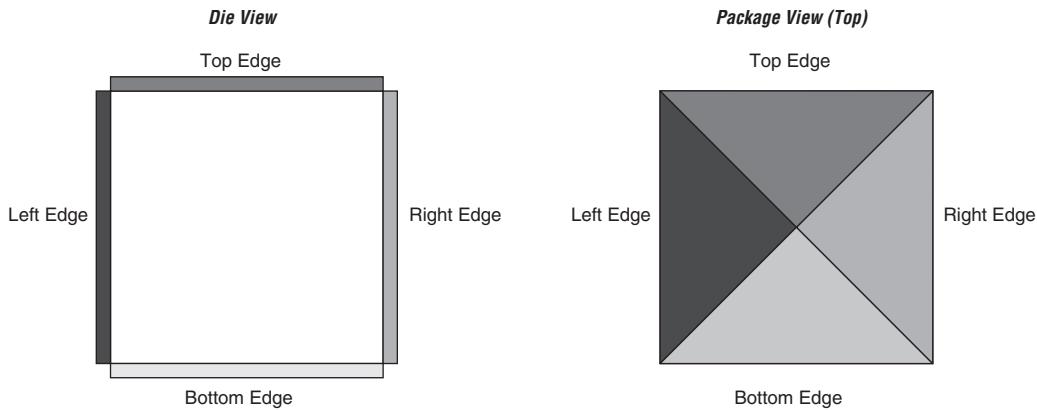
You can assign a location to your pins using the **Assignment Editor**. Choose **Assignment Editor** (Assignments menu) to open the Assignment Editor. Select the **Pins category** from the **Category** list. Type the pin name and select a location from the location list. For Stratix II, Stratix, Stratix GX, Cyclone II, and Cyclone devices, you can also assign a pin to an I/O bank or to an edge on the die.

It is common to place a group of pins (buses) with compatible I/O standards in the same bank. For example, two buses with two I/O standards, 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. You can also use edge-location assignments to improve circuit board routing ability of large buses, since they are close together near an edge. [Figure 5–5](#) shows the Altera device package edges.



You can also make pin assignments using Tcl commands or with the Timing Closure Floorplan. See the *Timing Closure Floorplan* chapter in Volume 2 of the *Quartus II Handbook*.

Figure 5–5. Die View and Package View of the Four Edges on an Altera Device

Suggested & Partial Placement

The **Start I/O Assignment Analysis** command automatically assigns locations to pins that do not have pin location assignments. For example, if you assign an edge location to a group of LVDS pins, the I/O assignment analysis assigns pin locations for each LVDS pin in the specified edge location and then performs legality checks.

Choose **Back-Annotate Assignments** (Assignments menu), select **Pin & device** assignments, and click **OK** to accept the suggested pin locations. Back-annotation saves your pin and device assignments in the QSF.

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 database.

To generate a mapped netlist, choose **Start > Start Analysis & Synthesis** (Processing menu). You can also use the **quartus_map** executable to run analysis and synthesis.

Type the following at a (non-Tcl) system command prompt:

```
quartus_map <project name> ↵
```

Understanding the I/O Assignment Analysis Report & Messages

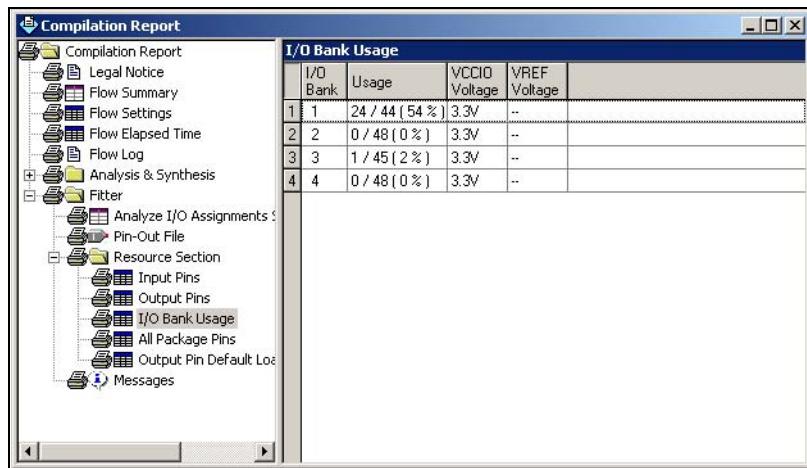
The **Start I/O Assignment Analysis** command generates a detailed analysis report (see [Figure 5–6](#)) and a Pin-out file (.pin). The detailed messages in the report help you quickly understand and resolve pin assignment errors. Each detailed message includes a related node name and a description of the problem.

You can view the report file by choosing **Compilation Report** (Project menu). The Fitter section of the Compilation Report contains the following five sections:

- Analyze I/O Assignment Summary
- Resource Section
- 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 5–6. Summary of the I/O Bank Usage in the I/O Assignment Analysis Report



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. Choose **Utility Windows > Messages** (View menu) to open the Messages window.

Use the location box to help resolve the error messages. Select from the location list and click **Locate**.

Figure 5–7. shows an example of error messages reported by I/O assignment analysis:

Figure 5–7. Error Message Report by I/O Assignment Analysis

```

[REDACTED]

```

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. 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 `quartus_sh --qhelp` at a system command prompt to run the Quartus II Command-Line and Tcl API Help browser.



For more information on Quartus II scripting support, including examples, refer to the *Tcl Scripting* and *Command-Line Scripting* chapters 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, see [page 5–11](#).

Tcl Command

Enter the following in a Tcl console or script:

```
execute_flow -check_ios
```

Command Prompt

Type the following at a (non-Tcl) system command prompt:

```
quartus_fit <project-name> --check_ios ↵
```

Reserving Pins

Use the following Tcl command to reserve a pin. For more information about reserving pins, see “[Reserving Pins](#)” on page 5–8.

```
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" and "AS SIGNAL PROBE 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, see “[Location Assignments](#)” on page 5–9.

```
set_location_assignment <location> -to <signal name>
```

Valid locations are pin location names, such as Pin_A3. The Stratix series products and Cyclone 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 up to IOBANK_n, where n is the number of I/O banks in a particular device.

Generating a Mapped Netlist

You can generate a mapped netlist with a Tcl command or with a command run at a command prompt. For more information about generating a mapped netlist, see [page 5–10](#).

Tcl Command

Enter the following in a Tcl console or script:

```
execute_module -tool map
```

The **execute_module** command is in the flow package.

Command Prompt

Type the following at a (non-Tcl) system command prompt:

```
quartus_map <project name> ↵
```

Conclusion

The **Start I/O Assignment Analysis** command quickly and thoroughly validates the legality of your pin-related assignments. This helps reduce development time by catching illegal pin assignments early in the design cycle without wasting long design compilations.

By providing the designer with more confidence in the device pin-outs at an early stage, board layout engineers can work in parallel with FPGA designers to achieve a time-to-market advantage.



Section III. Area Optimization & Timing Closure

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 many advanced design analysis tools that allow for detailed timing analysis of your design, including a fully integrated Timing Closure Floorplan Editor. With these tools and options, critical paths can be easily determined and located 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 6, Design Optimization for Altera Devices
- Chapter 7, Timing Closure Floorplan
- Chapter 8, Netlist Optimizations & Physical Synthesis
- Chapter 9, Design Space Explorer
- Chapter 10, LogicLock Design Methodology
- Chapter 11, Synplicity Amplify Physical Synthesis Support

Revision History

Chapter 11, *Timing Closure in HardCopy Devices* was removed from this *Quartus II Handbook*.

The table below shows the revision history for Chapter 6 to 11.

Chapter(s)	Date / Version	Changes Made
6	Dec. 2004 v2.1	Updated for Quartus II software version 4.2: <ul style="list-style-type: none"> ● Re-organized chapter. ● Added “Early Timing Estimation” segment. ● Removed “Incremental Fitting” segment. ● In “Optimization Advisors” segment, added an example with a screenshot. ● Added “Restructure Multiplexers” to the “Resource Utilization Optimization Techniques (LUT-Based Devices)” segment. ● Added the DSP Block Balancing logic option to the “Retarget or Balance DSP Blocks” segment. ● Added information about using wildcards in the “Duplicate Logic for Fan-Out Control” segment. ● Updated Figures 6–2, 6–3, 6–4, and 6–5. ● Added a technique at bottom of Table 6–5. ● Added Device Setting to Table 6–12.
	June 2004 v2.0	<ul style="list-style-type: none"> ● Updates to tables, figures. ● New functionality in the Quartus II software version 4.1.
	Feb. 2004 v1.0	Initial release.
7	Dec. 2004 v2.1	Updated for Quartus II software version 4.2: <ul style="list-style-type: none"> ● Removed By Delay and Show Routing Delays options from the “Viewing Critical Paths” segment. ● Updated figures.
	June 2004 v2.0	<ul style="list-style-type: none"> ● Updates to tables, figures. ● New functionality in the Quartus II software version 4.1.
	Feb. 2004 v1.0	Initial release.

Chapter(s)	Date / Version	Changes Made
8	Dec. 2004 v2.1	<p>Updated for Quartus II software version 4.2:</p> <ul style="list-style-type: none"> ● General formatting and editing updates. ● Additional description about fixed and primitive node names for synthesis netlist optimization and physical synthesis options on page page 8–2. ● Updated Figure 8–1. ● Clarified APEX support in “Gate-Level Register Retiming”. ● Added information about node name changes for atoms during physical synthesis on page 8–10. ● Deleted section on “Physical Synthesis Report.”
	June 2004 v2.0	<ul style="list-style-type: none"> ● Updates to tables and figures. ● New functionality in the Quartus II software version 4.1.
	Feb. 2004 v1.0	Initial release.
9	Dec. 2004 v2.1	<ul style="list-style-type: none"> ● Updates to tables and figures. ● New functionality in the Quartus II software version 4.2.
	June 2004 v2.0	<ul style="list-style-type: none"> ● Updates to tables and figures. ● New functionality in the Quartus II software version 4.1.
	Feb. 2004 v1.0	Initial release.
10	Dec. 2004 v2.2	<ul style="list-style-type: none"> ● Updates to tables and figures. ● New functionality in the Quartus II software version 4.2.
	August 2004 v2.1	<ul style="list-style-type: none"> ● New functionality in the Quartus II software version 4.1 Sp1.
	June 2004 v2.0	<ul style="list-style-type: none"> ● Updates to tables and figures. ● New functionality in the Quartus II software version 4.1.
	Feb. 2004 v1.0	Initial release.
11	Dec. 2004 v1.1	<ul style="list-style-type: none"> ● Chapter 11 was formerly Chapter 12. ● Updates to tables and figures. ● New functionality in the Quartus II software version 4.2
	Feb. 2004 v1.0	Initial release.

qii52005-2.1

Introduction

Techniques for achieving the highest design performance are important when designing for programmable logic devices (PLDs). The tools that facilitate these techniques must provide the highest level of flexibility without compromising ease-of-use. The optimization features available in the Quartus® II software allow you to meet performance requirements by facilitating optimization at multiple points in the design process. You can apply optimizations to an overall design or to sub-modules of a design that are integrated later.

This chapter explains techniques to reduce resource usage, improve timing performance, and reduce compilation times when designing with Altera® devices. It also explains how and when to use some of the Quartus II software features described in detail in other chapters of the *Quartus II Handbook*.

The results of following these recommendations are design-specific. Applying each technique may not always improve your results. Settings and options in the Quartus II software are set to default values that, on average, provide the best trade-off between compilation time, resource utilization, and timing performance. The software allows you to adjust these settings to concentrate on your area of interest and see if different settings provide better results for your specific design. Use the optimization flow described in this chapter to explore various compiler settings and determine the combination of techniques that provide the required results for your design.

Stages in the Compilation Process Described in this Chapter

The first stage in the optimization process is to perform an initial compilation (see “[Initial Compilation](#)” on page [6-3](#)) to establish a baseline that you can use to analyze your design. “[Design Analysis](#)” on page [6-8](#) explains how to analyze the compilation results, and provides links to the sections of this chapter where you can proceed with optimizing resources, performance, or compilation time. Altera recommends optimizing resource usage first, then I/O timing, then f_{MAX} timing, so this chapter presents the recommendations for each stage in that order. This chapter describes the analysis and optimization process for look-up table (LUT)-based devices, including FPGA devices and MAX® II device family CPLDs first. It then describes the process for macrocell-based CPLDs (in the MAX 7000 and MAX 3000 device

families). The final optimization section covers compilation time optimization, which is device independent. In addition to the user interface techniques described throughout the chapter, you can also use Tcl or command line scripting to apply the techniques. Scripting techniques are described in “[Scripting Support](#)” on page 6–64.

Design Space Explorer

You can use the Design Space Explorer (DSE) to run multiple compilations to try some of the techniques outlined in this chapter. The DSE is a utility that automates the process of 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.



For more information, refer to the *Design Space Explorer* chapter in Volume 2 of the *Quartus II Handbook*.

Optimization Advisors

The Quartus II software includes the **Resource Optimization Advisor** and the **Timing Optimization Advisor** (Tools menu) that provide guidance for making settings to optimize your design. The advisors cover many of the suggestions listed in this chapter. If you open the advisors after compilation, the Optimization Advisors show icons that indicate which resources or timing constraints were not met.

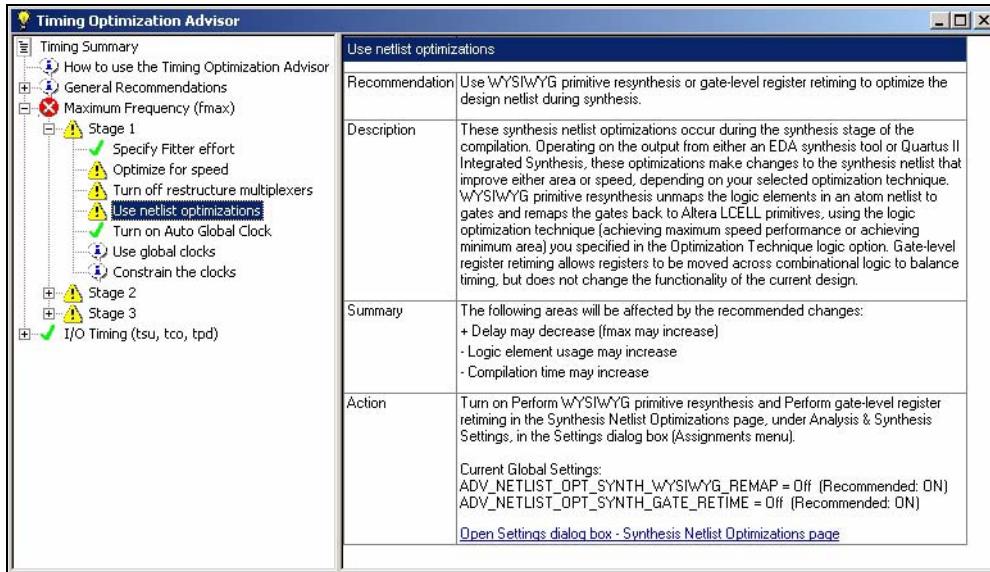
When you expand one of the categories, such as **Logic Element Usage** or **Maximum Frequency (f_{MAX})**, recommendations are split into stages. The stages show the order in which you should apply the recommended settings. The first stage contains the options that are easiest to change, make the least 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. Refer to the “How to use” page in the Advisor for a legend that describes each icon.

There is a link from each recommendation to the appropriate location in the Quartus II user interface where you can change the setting. This provides you with the most control over which settings are made, and helps you learn about the settings in the software.

The example in [Figure 6–1](#) shows the Timing Optimization Advisor after compiling a design that fails to meet its frequency requirements, so there is an error icon next to the **Maximum Frequency (f_{MAX})** entry in the advisor. The check mark icons in the list of recommendations for **Stage 1** indicate recommendations that are already followed in this project. The warning icons indicate suggestions where the recommended setting is not made appropriately in this project. The information icon indicates

more general suggestions, where the advisor does not report whether the recommendation has already been followed but it explains what you should do to meet the recommendations.

Figure 6–1. Timing Optimization Advisor



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 assignments made. This section describes the basic assignments and settings to make for your initial compilation.

Device Setting

Assigning a specific device determines the timing model that the Quartus II software uses during compilation. It is important to choose the correct speed grade to obtain accurate results and the best optimization. The device size and the package determines the device pin-out and how many resources are available in the device.

Choose the target device on the **Device** page of the **Settings** dialog box (Assignments menu).

Timing Requirements Settings

An important step in obtaining the highest performance, especially for high performance FPGA designs, is the application of detailed timing requirements. The Quartus II PowerFit™ Fitter attempts to meet or exceed specified timing requirements (depending on the selected options as described in “[Fitter Effort Setting](#)” on page 6–7). The Quartus II physical synthesis optimizations are also performed based on the constraints in specified timing requirements (see “[Synthesis Netlist Optimizations & Physical Synthesis Optimizations](#)” on page 6–32 for more information). In addition, timing requirements are used during timing analysis. The compilation report shows whether timing requirements were met and provides detailed timing information on paths that violate the timing requirements.

Make timing requirement settings in the **Timing Requirements & Options** page of the **Settings** dialog box (Assignments menu) or with the Assignment Editor. On the **Timing Requirements & Options** page, use the **Delay requirements**, **Minimum delay requirements**, and **Clock Settings** boxes to enter global requirements, or select **Settings for individual clock signals** to make settings on individual clocks (recommended for multiple-clock designs). First create the clock setting, then apply it to the clock node in the design. Running the **Timing Wizard** (Assignments menu) makes it easy to make individual clock settings by allowing you to create each clock setting and apply it to the appropriate clock node.

Every clock signal should have an accurate clock setting assignment. All I/O pins for which t_{SU} , t_H , or t_{CO} is to be optimized should also have settings. In addition, if you have t_{PD} or minimum t_{CO} constraints, those should be specified as well. Therefore, if there is more than one clock or there are different I/O requirements for different pins, use the **Settings** dialog box or the Timing Wizard to make multiple clock settings and the Assignment Editor to make individual I/O assignments rather than using the global settings.

It is important to make any complex timing assignments according to the needs of the design, including multicycle and cut-timing path assignments. This information allows the Quartus II software to make appropriate trade-offs between paths. Make these settings with the Assignment Editor.



When there are timing constraints in the design, the Quartus II software does not attempt to optimize clocks that are unconstrained. Wherever possible, specify timing constraints on all clock signals in the design for best results.



For more information on how to make timing assignments, refer to the *Quartus II Timing Analysis* chapter in Volume 3 of the *Quartus II Handbook*. Also see Quartus II Help.

Smart Compilation Setting

Smart compilation can reduce compilation time, especially when you have multiple compilation iterations during the optimization phase of the design process; however, it will use more disk space. Turn on the **Use Smart compilation** option on the **Compilation Process Settings** page of the **Settings** dialog box (Assignments menu).

Early Timing Estimation

The Quartus II software provides an **Early Timing Estimate** feature that allows you to get an estimate of your design's timing results before performing full placement and routing. This feature runs the fitter up to 10 times faster than a full fit and generates a full timing report based on estimated delays for the design. The fit is not fully optimized or routed, and therefore the timing report is only an estimate. Typically, the estimated delays are within 20% of what a full compilation can achieve.

You can modify the type of timing estimation used by this feature on the **Early Timing Estimate** page under **Compilation Process Settings** in the **Settings** dialog box (Assignments menu). Using the **Optimistic** option generates timing estimates that are unlikely to be exceeded by a full compilation, while the **Pessimistic** option generates timing estimates that are likely to be exceeded by a full compilation. The **Realistic** option, which is the default, generates delay estimates that will likely be closest to a full compilation's results.

To use this command to generate your initial compilation results, choose **Start > Start Early Timing Estimate** (Processing menu) after you have performed analysis and synthesis in the Quartus II software. In cases where you want a quick estimate of your design results before proceeding with further design or synthesis tasks, this command can save you significant compilation time.

Timing-Driven Compilation Settings

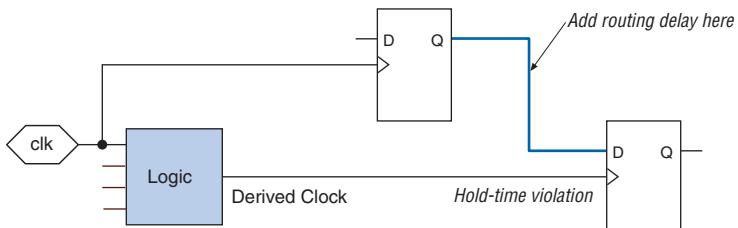
Ensure that the **Optimize timing** and the **Optimize I/O cell register placement for timing** options on the **Fitter Settings** page of the **Settings** dialog box (Assignments menu) are set appropriately. Turning on these options allows the Quartus II software to optimize your design based on the timing requirements that you have specified with various timing assignments.

The **Optimize hold timing** option is another timing-driven compilation option that directs the Quartus II software to optimize minimum delay timing constraints. This option is available only for Stratix® II, Stratix, Stratix GX, Cyclone™ II, Cyclone, and MAX II devices. When this option is turned on, the Quartus II software adds delay to connections as needed to guarantee that the minimum delays required by these constraints are satisfied.

By default, the Fitter optimizes these constraints using the worst case timing model, which uses the worst case or slowest timing delay numbers. If you turn on **Optimize fast-corner timing** on the **Fitter Settings** page of the **Settings** dialog box (Assignments menu), the Fitter also analyzes the fast delay corner, which uses the best case or fastest timing numbers. Using the two different timing models accounts for process, voltage, and temperature variations for each device.

If you choose **IO Paths and Minimum TPD Paths** (the default choice under **Optimize hold timing**), the Fitter works to meet hold times (t_H) from device input pins to registers, minimum delays from I/O pins or registers to I/O pins or registers (t_{PD}), and 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 6–2](#), 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, this is not the recommended way to fix internal hold violation problems. Altera recommends instead that you fix internal register to register hold problems by making changes to your design, such as using a clock enable instead of a derived or gated clock.

Figure 6–2. Optimize Hold Timing Option Fixing an Internal Hold Time Violation



For good design practices that can help eliminate internal hold time violations, see the *Design Recommendations for Altera Devices* chapter in Volume 1 of the *Quartus II Handbook*.

Fitter Effort Setting

You can change the **Fitter Effort** setting on the **Fitter Settings** page of the **Settings** dialog box. The default setting in the Quartus II software depends on the device family specified.

The **Standard Fit** option attempts to exceed specified timing requirements and achieve the best possible timing results for your design. This Fitter effort setting usually involves the longest compilation time.

The **Fast Fit** option reduces the amount of optimization effort for each algorithm employed during fitting. This reduces the compilation time by about 50%, while resulting in a fit that has, on average, 10% lower f_{MAX} than that achieved using the **Standard Fit** setting. For a small minority of hard-to-fit circuits, the reduced optimization resulting from using the **Fast Fit** option can result in the first fitting attempt being unrouteable, resulting in multiple fitting attempts and a long fitting time.

The **Auto Fit** option (available only for Stratix II, Stratix, Stratix GX, Cyclone II, Cyclone, and MAX II devices) decreases compilation time by directing the Fitter to reduce Fitter effort after meeting the design's timing requirements if it meets internal routability requirements. The internal routability requirements reduce the possibility of routing congestion and help ensure quick, successful routing. If you want the Fitter to try to exceed the timing requirements by a certain margin before reducing Fitter effort, you can specify a minimum slack that the Fitter must try to achieve before reducing Fitter effort in the **Desired worst case slack** box. The **Auto Fit** option also causes the Quartus II Fitter to optimize for shorter compilation times instead of maximum performance when there are no timing constraints. For designs with no timing requirements, the resulting f_{MAX} is an average of 15% lower than 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 easy or no timing requirements, the **Auto Fit** option reduces compilation time by 40% on average.



Note that selecting this option does not guarantee that the Fitter meets the design's timing requirements, and specifying a minimum slack does not guarantee that the Fitter will achieve the slack.

I/O Assignments

The I/O standards and drive strengths specified for a design affect I/O timing. Specify these 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, Altera recommends leaving pin locations unconstrained to achieve the best results. If your pin locations are already fixed, make the pin assignments in the Quartus II software to constrain the compilation appropriately. “[Resource Utilization Optimization Techniques \(Macrocell-Based CPLDs\)](#)” on page 6–46 includes recommendations for making pin assignments, since your pin assignments can have a larger effect on your quality of results in smaller macrocell-based architectures.

You can assign I/O standards and pin locations with the **Assignment Editor** (Assignments menu).



For more information on I/O standards and pin constraints, see the appropriate device data sheet or handbook. For information on using the Assignment Editor, refer to the *Assignment Editor* chapter in Volume 2 of the *Quartus II Handbook*.

Design Analysis

The initial compilation establishes whether the design achieves a successful fit and meets the specified performance. The Compilation Report reports the design results. This section describes how to analyze your design results, which is the first stage in the design optimization process.

After design analysis, proceed to the other optimization stages, as follows.

For LUT-based devices (FPGAs and MAX II CPLDs):

- If your design does not fit, see “[Resource Utilization Optimization Techniques \(LUT-Based Devices\)](#)” on page 6–14 before trying to optimize I/O timing or f_{MAX} timing
- If the I/O timing performance requirements are not met, see “[I/O Timing Optimization Techniques \(LUT-Based Devices\)](#)” on page 6–24 before trying to optimize f_{MAX} timing
- If f_{MAX} performance requirements are not met, see “[f_{MAX} Timing Optimization Techniques \(LUT-Based Devices\)](#)” on page 6–31

For macrocell-based devices (MAX 7000 and MAX 3000 CPLDs):

- If your design does not fit, see “[Resource Utilization Optimization Techniques \(Macrocell-Based CPLDs\)](#)” on page 6–46 before trying to optimize I/O timing or f_{MAX} timing
- If the timing performance requirements are not met, see “[Timing Optimization Techniques \(Macrocell-Based CPLDs\)](#)” on page 6–54

For techniques to reduce the compilation time, see “[Compilation-Time Optimization Techniques](#)” on page [6–60](#).

Resource Utilization

Determining device utilization is important regardless of whether a successful fit is achieved. If your compilation results in a no-fit error, then resource utilization information is important to analyze 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 any other design changes could introduce fitting difficulties.

To determine resource usage, see the **Flow Summary** section of the Compilation Report. This section reports how many pins are used, as well as other device resources such as memory bits, digital signal processing (DSP) block 9-bit elements, 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** (Processing menu).



Note that for Stratix II devices, a device with low utilization does not have the lowest adaptive logic module (ALM) utilization possible. For Stratix II devices, the Fitter uses adaptive look-up tables (ALUTs) in different ALMs even when the logic could be placed within one ALM. The Quartus II Fitter works to get the best timing and routability results, which in general involves some spreading of 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.

If resource usage is reported as less than 100% and a successful fit was not achieved, then it is likely that there were not enough routing resources or that some assignments were illegal. In either case, a message appears in the **Processing** tab of the **Messages** window to explain the problem.

If the Fitter finishes very quickly, then a resource may be over-utilized or there may be an illegal assignment (an error message is also reported for illegal assignments). If the Quartus II software runs for a long time, then it is likely that a legal placement or route cannot be found. Look for compilation messages that give an indication of the problem.

You can use the Timing Closure Floorplan to view areas of routing congestion.



For details on using the Timing Closure Floorplan, see the *Timing Closure Floorplan* chapter in Volume 2 of the *Quartus II Handbook*.

I/O Timing (Including t_{PD})

To determine whether I/O timing has been met, see the **Timing Analyzer** section of the **Compilation Report** (Processing menu). The t_{SU} , t_H , and t_{CO} reports list the I/O paths, along with the “Required” timing number if you have made a timing requirement, its “Actual” timing number for the parameter as reported by the Quartus II software, and the slack, or difference between your requirement and the actual number as specified by the Quartus II software. If you have any point-to-point propagation delay assignments (t_{PD}), the t_{PD} report lists the corresponding paths.

The I/O paths that have not met the required timing performance are reported as having negative slack and are displayed in red, as shown in [Figure 6–3](#). Even if you have not made an I/O timing assignment on that pin, the “Actual” number is the timing number that you must meet for that parameter when the device runs in your system.

Figure 6–3. I/O Timing Analyzer Report

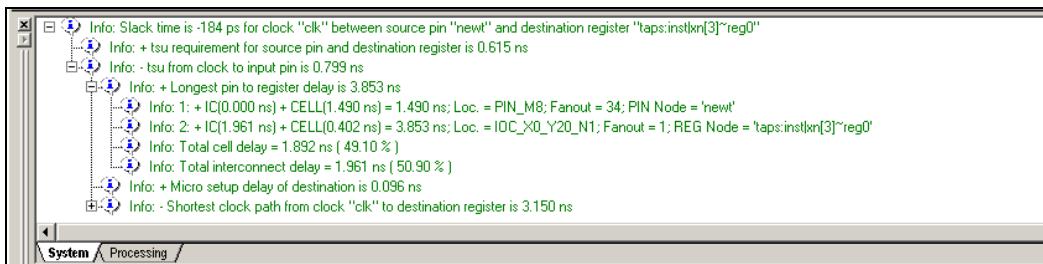
Slack	Required	Actual	From	To	To Clock
-0.184 ns	0.615 ns	0.799 ns	newt	taps:inst xn[1]~reg0	clk
-0.184 ns	0.615 ns	0.799 r			
-0.144 ns	0.615 ns	0.759 r			
-0.123 ns	0.615 ns	0.739 r			
-0.123 ns	0.615 ns	0.738 r			
0.112 ns	0.615 ns	0.503 r			
0.112 ns	0.615 ns	0.503 r			
0.470 ns	0.615 ns	0.145 r			
0.470 ns	0.615 ns	0.145 r			
0.470 ns	0.615 ns	0.145 r			
0.470 ns	0.615 ns	0.145 r			
0.470 ns	0.615 ns	0.145 ns	newt	taps:inst xn_2[2]~reg0	clk
0.470 ns	0.615 ns	0.145 ns	newt	taps:inst xn_3[2]~reg0	clk
0.489 ns	0.615 ns	0.126 ns	newt	taps:inst xn_2[6]~reg0	clk
0.489 ns	0.615 ns	0.126 ns	newt	taps:inst xn_3[6]~reg0	clk
0.489 ns	0.615 ns	0.126 ns	newt	taps:inst xn_2[4]~reg0	clk

To analyze the reasons that your timing requirements were not met, right-click a particular entry in the report and choose **List Paths** (as shown in [Figure 6–3](#)). A message listing the paths appears in the **System** tab of the **Messages** window. You can expand a selection, as shown in

Figure 6–4, by clicking the “+” icon at the beginning of the line. This is a good method of determining where along the path the greatest delay is located.

The List Paths report lists the slack time and how that slack time was calculated. By expanding the different 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 6–4. I/O Slack Report



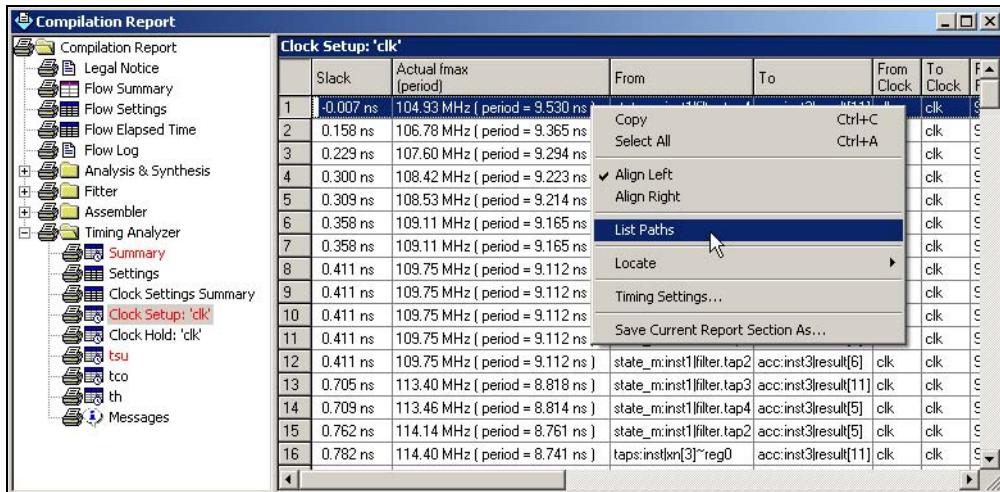
To visually analyze I/O timing, right-click on an I/O entry in the report and select **Locate > Locate in Timing Closure Floorplan** (right button pop-up menu) to highlight the I/O path on the floorplan. Negative slack indicates paths that failed to meet their timing requirements. There are also options to allow you to see all the intermediate nodes (that is, combinational logic cells) on a path and the delay for each level of logic. You can also look at the fan-in and fan-out of a selected node.



For more information on how timing numbers are calculated, refer to the *Quartus II Timing Analysis* chapter in Volume 3 of the *Quartus II Handbook*. For details on using the Timing Closure Floorplan, see the *Timing Closure Floorplan* chapter in Volume 2 of the *Quartus II Handbook*.

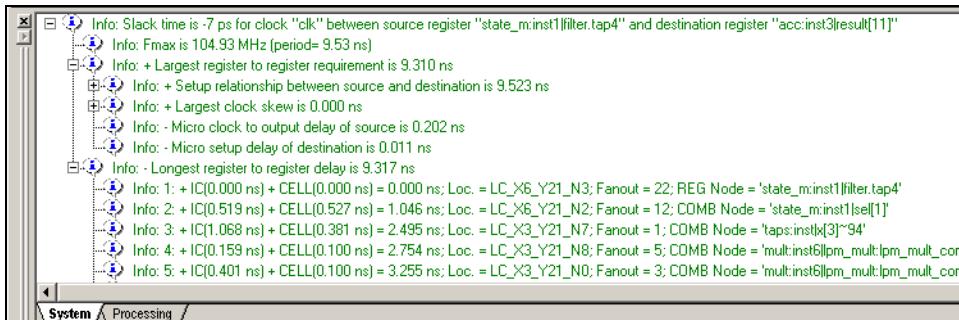
f_{MAX} Timing

To determine whether f_{MAX} timing requirements are met, see the **Timing Analyzer** section of the **Compilation Report** (Processing menu). The **Clock Setup** folder gives figures for the actual register-to-register f_{MAX} for each clock, as reported by the Quartus II software, and the slack, or difference between the timing requirement you have specified and the actual number specified by the Quartus II software. The paths that do not meet timing requirements are shown with a negative slack and appear in red (see **Figure 6–5**).

Figure 6–5. f_{MAX} Timing Analysis Report

To analyze why your timing requirements were not met, right click on a particular entry in the report and choose **List Paths** (as shown in [Figure 6–5](#)). A message listing the paths appears in the **System** tab of the **Messages** window. You can expand a selection, as shown in [Figure 6–6](#), by clicking the “+” icon at the beginning of the line. This is a good method of determining where along the path the greatest delay is located.

The List Paths report lists the slack time and how that slack time was calculated. By expanding the different 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 6–6. f_{MAX} Slack Report

You can visually analyze f_{MAX} paths by right-clicking on a path in the report and selecting **Locate > Locate in Timing Closure Floorplan** (right button pop-up menu) to display the **Timing Closure Floorplan**, which then highlights the path. Use the **Critical Path Settings** to select which failing paths to show. Turn them on or off with the **Show Critical Paths** command.

For more information on how timing analysis results are calculated, refer to the *Quartus II Timing Analysis* chapter in Volume 3 of the *Quartus II Handbook*. For details on using the Timing Closure Floorplan, see the *Timing Closure Floorplan* chapter in Volume 2 of the *Quartus II Handbook*.

You can also see a visual representation of 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 and choose **Locate > Locate in RTL Viewer** or **Locate > Locate in Technology Map Viewer** (right button pop-up menu). When you locate a timing path in the Technology Map Viewer, the viewer annotates the schematic with the same delay information as is presented when you use the List Paths command.

For more information on the netlist viewers, refer to the *Analyzing Designs with the Quartus II RTL Viewer & Technology Map Viewer* chapter in Volume 1 of the *Quartus II Handbook*.

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 includes Physical Synthesis if you have turned on those options. The **Flow Elapsed Time** section of the **Compilation Report** shows how much time the Analysis & Synthesis and Fitter modules took. The **Fitter Messages** report in the **Fitter** section of the **Compilation Report** shows how much time was spent in placement and how much time was spent in routing.

-  The applicable messages say Info: Fitter placement operations ending: elapsed time = <n> seconds and Info: Fitter routing operations ending: elapsed time = <n> seconds.

Placement describes the process of finding optimum locations for the logic in your design. Routing describes 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 easy to route.

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 f_{MAX} timing optimization. First, ensure that you have set the basic constraints described in “[Initial Compilation](#)” on page 6–3. If a design is not fitting into a specified device, use the techniques in this section to achieve a successful fit.

Resolving Resource Utilization Issues Summary

[Table 6–1](#) shows recommendations to reduce resource utilization and the recommended order in which to try the options, starting with those requiring the least effort and having the greatest effect.

-  The Design Space Explorer (DSE) Tcl/Tk script can be used to automate successive compilations of a design, each employing different options.

Once resource utilization has been optimized and your design fits in the desired target device, you can proceed to optimize I/O timing, as described in the “[I/O Timing Optimization Techniques \(LUT-Based Devices\)](#)” section.

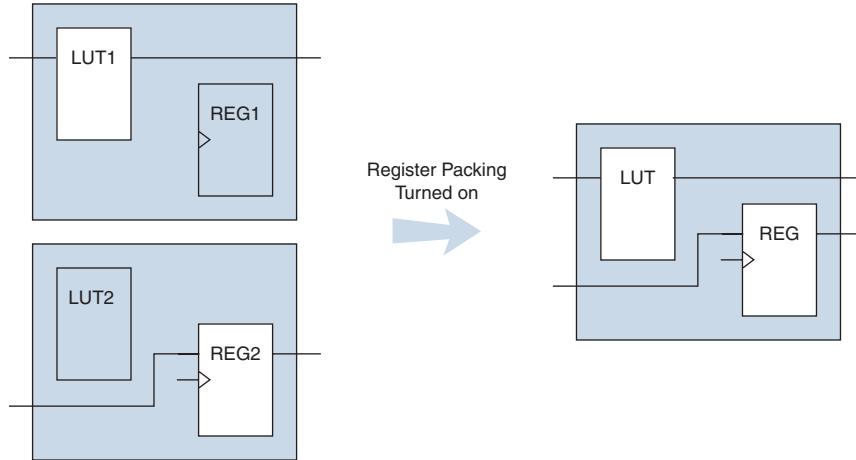
Table 6–1. Techniques for Resolving Resource Utilization Issues

Issue	Design Options to Employ
Too many logic cells used or logic cells do not fit	1. Use register packing (page 6–16) 2. Remove Fitter constraints (page 6–19) 3. Perform WYSIWYG primitive resynthesis (page 6–19) 4. Optimize synthesis for area (page 6–20) 5. Change state machine encoding (page 6–20) 6. Flatten the hierarchy (page 6–21) 7. Restructure multiplexers (page 6–21) 8. Optimize source code (page 6–23) 9. Use a larger device (page 6–24)
Too many memory blocks used	1. Retarget memory blocks (page 6–22) 2. Remove Fitter constraints (page 6–19) 3. Optimize source code (page 6–23) 4. Use a larger device (page 6–24)
Too many DSP blocks used	1. Retarget or balance DSP blocks (page 6–22) 2. Remove Fitter constraints (page 6–19) 3. Optimize source code (page 6–23) 4. Use a larger device (page 6–24)
Problems placing I/O pins	1. Change pin assignments (page 6–24) 2. Use a larger package with the same device density (page 6–24) 3. Use a larger device density with a larger pin count (page 6–24)
Too many routing resources used	1. Remove Fitter constraints (page 6–19) 2. Optimize synthesis for area (page 6–20) 3. Change state machine encoding (page 6–20) 4. Flatten the hierarchy (page 6–21) 5. Restructure multiplexers (page 6–21) 6. Optimize source code (page 6–23) 7. Use a larger device (page 6–24)

Use Register Packing

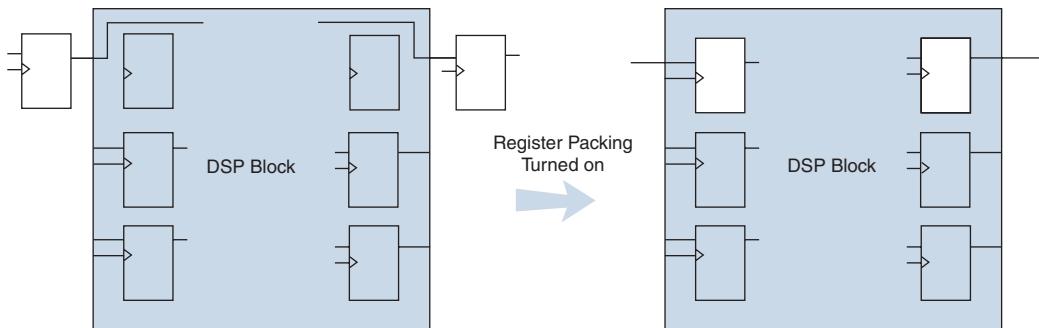
The **Auto Packed Registers** option is available regardless of the tool used to synthesize the design. Register packing combines a logic cell where only the register is used with another logic cell where only the lookup table (LUT) is used, and implements both functions in a single logic cell. Figure 6–7 shows the packing and the gain of one logic cell.

Figure 6–7. Register Packing



Registers may also be packed into DSP blocks as shown in Figure 6–8.

Figure 6–8. Register Packing in DSP Blocks



The following list indicates the most common cases in which register packing can help 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**—Default setting 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 Stratix II, Stratix, Stratix GX, Cyclone II, Cyclone, and MAX II devices.
- **Auto**—Attempts 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 are deemed to 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 Stratix II, Stratix, Stratix GX, Cyclone II, Cyclone, and MAX 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. To turn on register packing, turn on the **Auto Packed Registers** option by clicking **More Settings** on the **Fitter Settings** page of the **Settings** dialog box (Assignments menu).

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 6–2](#) shows typical results for Stratix II devices, [Table 6–3](#) shows typical results for Cyclone II devices, and [Table 6–4](#) shows typical results for Stratix, Stratix GX, and Cyclone devices.

Note that the **Auto** setting performs more aggressive register packing as needed, so the typical results vary depending on the device logic utilization.

Table 6–2. Typical Register Packing Results for Stratix II Devices

Register Packing Setting	Relative f_{MAX}	Relative LE Count
Off	0.95	1.29
Normal	1.00	1.00
Minimize Area	0.98	0.97
Minimize Area with Chains	0.98	0.97
Auto (default)	1.0 until device is very full, then gradually to 0.98 as required	1.0 until device is very full, then gradually to 0.97 as required

Table 6–3. Typical Register Packing Results for Cyclone II Devices

Register Packing Setting	Relative f_{MAX}	Relative LE Count
Off	0.97	1.40
Normal	1.00	1.00
Minimize Area	0.96	0.93
Minimize Area with Chains	0.94	0.91
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 0.91 as required

Table 6–4. Typical Register Packing Results for Stratix, Stratix GX & 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 or difficult-to-meet constraints may not fit the targeted device. This can occur when the location or LogicLock™ assignments are too strict and there are not enough routing resources.

In this case, use the **Routing Congestion** view in the **Timing Closure Floorplan** 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 Timing Closure Floorplan. You can remove LogicLock assignments in the Timing Closure Floorplan as well, or use the **LogicLock Regions Window** (Assignments menu). You can also choose **Remove Assignments** (Assignments menu) and turn on the assignment categories you want to remove from the design in the **Available assignment categories** list.



For more information on the **Routing Congestion** view in the **Timing Closure Floorplan**, see the Quartus II Help. Note that the feature is available only when you have chosen to use the **Field View** (View menu).

Perform WYSIWYG Resynthesis for Area

If you use another EDA synthesis tool and wish to see if the Quartus II software can re-map the circuit so that fewer LEs or ALMs are used, perform the following steps:

1. Turn on **Perform WYSIWYG primitive resynthesis (using optimization techniques specified in Analysis & Synthesis settings)** on the **Synthesis Netlist Optimizations** page under **Analysis & Synthesis Settings** in the **Settings** dialog box (Assignments menu), or apply the **Perform WYSIWYG Primitive Resynthesis** logic option to a specific module in your design with the **Assignment Editor** (Assignments menu).
2. Choose **Area** for **Optimization Technique** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu), or set the **Optimization Technique** logic option to **Area** for a specific module in your design with the **Assignment Editor** (Assignments menu).
3. Recompile the design.



Performing WYSIWYG resynthesis for **Area** in this way typically reduces f_{MAX} .

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, as described in this section.

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 may result in higher device resource usage if the constraints are too aggressive.



For information on setting timing requirements and synthesis options in other synthesis tools, see the appropriate chapter in the *Synthesis* section in Volume 1 of the *Quartus II Handbook*, or your synthesis software's documentation.

If device utilization is an important concern, some synthesis tools offer an easy way to optimize for area instead of speed. If you are using the Quartus II integrated synthesis, choose **Area for Optimization Technique** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu). 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 (potentially at the expense of f_{MAX} 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**. In some synthesis tools, not specifying an f_{MAX} requirement may result in less logic utilization. Other attributes or options may also be available to help improve the quality of results, including the recommendations in the following paragraphs.

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 device utilization. The effect of state machine encoding differs 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. In the Quartus II integrated synthesis, choose **Minimal Bits for State Machine Processing**

on the **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu). You can also specify this logic option for specific modules or state machines in your design with the Assignment Editor.

Flatten the Hierarchy During Synthesis

Synthesis tools typically provide you with the option of preserving hierarchical boundaries, which may be useful for verification or other purposes. Optimizing across hierarchical boundaries, however, allows the synthesis tool to perform the most logic minimization, which may reduce area. Therefore, flatten your design hierarchy whenever possible to achieve best results. If you are using the Quartus II integrated synthesis, ensure that the **Preserve Hierarchical Boundary** logic option is turned off (that is, ensure that you have not turned on the option in the Assignment Editor or with Tcl assignments).

Restructure Multiplexers

Multiplexers form a large portion of the logic utilization in many FPGA designs. By optimizing your multiplexing logic, you can achieve a more efficient implementation in your Altera device.



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*.

The Quartus II software provides the **Restructure Multiplexers** logic option, available on the **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu), which can 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. The option may negatively affect your design's f_{MAX} . This option is turned on automatically when you set the Quartus II **Analysis & Synthesis Optimization Technique** option to **Area**, which can be set globally on the **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu).



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*.

Retarget Memory Blocks

If the design fails to fit because it runs out of device memory resources, it may be due to a lack of a certain type of memory. For example, a design may require two M-RAM blocks and be targeted for a Stratix EP1S10 device, which has only one. By building one of the memories with a different size memory block, such as an M4K memory block, it may be possible to obtain a fit.

If the memory was created with the MegaWizard® Plug-In Manager, simply open the MegaWizard Plug-In Manager and edit the RAM block type so that it targets a new memory block size.

ROM and RAM memory blocks can also be inferred from your hardware description language (HDL) code, and your synthesis software may place large shift registers into memory blocks with the `altshift_taps` megafunction. This inference can be turned off in your synthesis tool so that the memory is placed in logic instead of in memory blocks. In Quartus II integrated synthesis, disable inference by turning off the **Auto RAM Replacement**, **Auto ROM Replacement**, or **Auto Shift Register Replacement** logic option as appropriate for your whole project on the **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu), or by disabling the option for a specific block in the Assignment Editor.

Depending on your synthesis tool, you may be able to set the RAM block type for inferred memory blocks as well. In Quartus II integrated synthesis, set the `ramstyle` attribute to the desired memory type for the inferred RAM blocks: M512, M4K, or M-RAM.



For more information on memory inference control, see the appropriate chapter in the *Synthesis* section in Volume 1 of the *Quartus II Handbook*, or your synthesis software's documentation.

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, making it possible to retarget some of the DSP blocks to logic to obtain a fit.

If the DSP function was created with the MegaWizard Plug-In Manager, simply open the MegaWizard Plug-In Manager and edit the block so it targets logic instead of DSP blocks. Megafunctions use the `DEDICATED_MULTIPLIER_CIRCUITRY` parameter to control the implementation.

DSP blocks can also be inferred from your HDL code from multipliers, multiply-adders, and multiply-accumulators. This inference can be turned off in your synthesis tool. In Quartus II integrated synthesis, disable inference by turning off the **Auto DSP Block Replacement** logic option for your whole project on the **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu), or by disabling the option for a specific block with the Assignment Editor.



For more information on disabling DSP block inference in other synthesis tools, see 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 enables a process during synthesis that implements DSP block slices in logic cells or in different DSP block modes. The default **Auto** setting allows DSP block balancing to automatically convert the DSP block slices as appropriate to minimize the area and maximize the speed of the design. You can use other settings of the option for a specific node or entity, or project-wide, 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 on 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 in the design, modify the design at the source to achieve the desired results. You may be able to improve logic efficiency by making design-specific changes to your source code. In many cases, optimizing the design's source code can have a significant effect on your logic utilization.

If your design does not fit because of logic resources, but you have unused memory or DSP blocks, check whether you have code blocks in your design that describe memory or DSP functions but 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.



For coding style guidelines including examples of HDL code for inferring memory and DSP functions and other coding examples, refer to the *Recommended HDL Coding Styles* chapter in Volume 1 of the *Quartus II Handbook*.

Modify Pin Assignments or Choose a Larger Package

If a design with pin assignments fails to fit, try compiling the design without the pin assignments to see whether a fit is possible for the design in the specified device and package. You can also try this approach if a Quartus II error message indicates fit problems due to pin assignments.

If the design fits when all pin assignments are ignored or when several pin assignments are ignored or moved, it may be necessary 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 with more available user I/O pins.

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.

I/O Timing Optimization Techniques (LUT-Based Devices)

The next stage of design optimization focuses on I/O timing. Ensure that you have made the appropriate assignments as described in “[Initial Compilation](#)” on page [6-3](#), and that the resource utilization is satisfactory, before proceeding with I/O timing optimization. Because changes to the I/O path affect the internal f_{MAX} , complete this stage before proceeding to the f_{MAX} timing optimization stage.

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 & Clock-to-Output Times Summary

Table 6–5 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. Keep in mind that reducing t_{SU} times increases hold (t_H) times.

Table 6–5. Improving Setup & Clock-to-Output Times Note (1)

Technique	t_{SU}	t_{CO}
Ensure that the appropriate constraints are set for the failing I/Os (page 6–7)	✓	✓
Use timing-driven compilation for I/O (page 6–25)	✓	✓
Use fast input register (page 6–26)	✓	
Use fast output register and fast output enable register (page 6–26)		✓
Set Decrease Input Delays to Input Register = ON or decrease the value of Input Delay from Pin to Input Register (page 6–27)	✓	
Set Decrease Input Delays to Internal Cells = ON or decrease the value of Input Delay from Pin to Internal Cells (page 6–27)	✓	
Set Increase Delay to Output Pin = OFF or decrease the value of Delay from Output Register to Output Pin (page 6–27)		✓
Increase the value of Input Delay from Dual-Purpose Clock Pin to Fan-Out Destinations (page 6–29)	✓	
Use PLLs to shift clock edges (page 6–29)	✓	✓
Use the Fast Regional Clock option (page 6–30)		✓
Change how hold times are optimized for MAX II devices (page 6–30)	✓	

Note to Table 6–5:

- (1) These options may not apply for all device families.

Once I/O timing has been optimized, you can proceed to optimize f_{MAX} , as described in the “ f_{MAX} Timing Optimization Techniques (LUT-Based Devices)” section.

Timing-Driven Compilation

Perform I/O timing optimization using the **Optimize I/O cell register placement for timing** assignment located on the **Fitter Settings** page of the **Settings** dialog box (Assignments menu). 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 on 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 I/O cell register placement for timing** option only affects pins that have a t_{SU} or t_{CO} requirement. Using the I/O register is only possible if the register directly feeds a pin or is fed directly by a pin. This setting does not affect registers with 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 port in APEX 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 above are optimized using the regular Quartus II Fitter optimizations.

Fast Input, Output & 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. 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 on 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 I/O cell register placement for timing** option, located on the **Fitter Settings** page of the **Settings** dialog box (Assignments menu), 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 I/O cell register placement for timing** option is turned on.

The three fast I/O options (**Fast Input Register**, **Fast Output Register**, and **Fast Output Enable Register**) can also 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 Stratix II, Stratix, Stratix GX, Cyclone II, Cyclone, and MAX II devices, the Quartus II software automatically adjusts the applicable programmable delays to help meet timing requirements. For the APEX families of devices, the default values are set to generally avoid any hold time problems. Programmable delays are advanced options that should be used only after you have compiled a project, checked the I/O timing, and determined that the timing is unsatisfactory. For detailed information on the effect of these options, see the device family handbook or data sheet.

Assign programmable delay options to supported nodes with the Assignment Editor.

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 Quartus II Compilation Report.

You can also view and modify the delay chain setting for the target device with the Quartus II Chip Editor and Resource Property Editor. When you use the Resource Property Editor to make changes after performing a full compilation, you don't need to recompile the entire design; you can save changes directly to the netlist. As these changes are made directly to the netlist, the changes are not made again automatically when you recompile the design. There are change management features to allow you to reapply the changes on subsequent compilations.



For more information on using the Quartus II Chip Editor and Resource Property Editor, refer to the *Design Analysis and Engineering Change Management with Chip Editor* chapter in Volume 3 of the *Quartus II Handbook*.

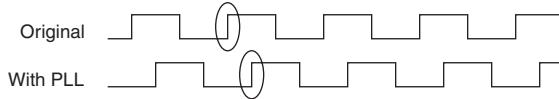
Table 6–6 summarizes the programmable delays available for Altera devices.

Table 6–6. Programmable Delays for Altera Devices (Part 1 of 2)			
Programmable Delay	Description	I/O Timing Impact	Device Families
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 input/bidirectional pin or register it feeds.	Decreases t_{SU} Increases t_H	Stratix, Stratix GX, Cyclone, APEX II, APEX 20KE, APEX 20KC, Mercury™, MAX 7000B
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 input/bidirectional pin.	Changes t_{SU} Changes t_H	Stratix II, 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 input/bidirectional pin or register it feeds.	Decreases t_{SU} Increases t_H	Stratix, Stratix GX, Cyclone, APEX II, APEX 20KE, APEX 20KC, Mercury, FLEX 10K®, FLEX® 6000, ACEX® 1K
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 a input or bidirectional pin.	Changes t_{SU} Changes t_H	Stratix II, 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 input/bidirectional pin or register it feeds.	Decreases t_{PD}	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 output/bidirectional pin or register feeding it.	Increases t_{CO}	Stratix, Stratix GX, APEX II, Mercury
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}	Stratix II

Table 6–6. Programmable Delays for Altera Devices (Part 2 of 2)			
Programmable Delay	Description	I/O Timing Impact	Device Families
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}	Stratix II, 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 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 should improve I/O timing automatically. If the timing requirements are still not met, most devices allow the PLL 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 the t_{SU} , while shifting it forward gives a better t_{SU} at the expense of t_{CO} and t_H . See [Figure 6–9](#). This technique can be used only in devices that offer PLLs with the phase shift option.

Figure 6–9. Shift Clock Edges Forward to Improve t_{SU} at the Expense of t_{CO} 

Note that 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 6–6](#).

Use Fast Regional Clocks in Stratix Devices

Stratix EP1S25, EP1S20, and EP1S10 devices and Stratix GX EP1SGX10 and EP1SGX25 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 can feed these clock nets directly. Fast regional clocks have less delay to I/O elements than regional or global clocks and are used for high fan-out control signals. Placing clocks on fast regional 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** logic option to control how the hold times, t_H , are optimized by the Quartus II software.

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 setting is **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 setting is **When tsu and tpd constraints permit**, the Fitter will guarantee 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 setting is **Off**, designs are optimized to meet user timing assignments only.

By setting this option to **Off** or **When tsu and tpd constraints permit**, you may be able to improve t_{SU} at the expense of t_H . This option can be set in the **Assignment Editor** (Assignments menu) or the **Analysis & Synthesis Settings** page or the **Fitter Settings** page of the **Settings** dialog box (Assignments menu).

f_{MAX} Timing Optimization Techniques (LUT-Based Devices)

The next stage of design optimization is to improve the f_{MAX} timing. There are a number of options available if the performance requirements are not achieved after compiling with the Quartus II software.



It is important to understand your design and apply appropriate assignments to increase performance. It is possible to decrease performance if assignments are applied without full understanding of the design or the effect of the assignments.

Improving f_{MAX} Summary

The choice of options and the adjustment of settings to improve f_{MAX} depends on the failing paths in the design. To achieve the best results relative to your performance requirements, apply the following options, compiling after each:

1. Apply netlist optimization options (including physical synthesis) ([page 6-32](#)).
2. Modify the seed ([page 6-33](#)). (This step may be omitted if a large number of critical paths are failing, or if paths are failing by large amounts.)
3. Apply synthesis options to optimize for speed:
 - Optimize synthesis for speed instead of area ([page 6-34](#))
 - Flatten the hierarchy ([page 6-35](#))
 - Set the synthesis effort to high ([page 6-35](#))
 - Change state machine encoding ([page 6-36](#))
 - Duplicate logic for fan-out control ([page 6-36](#))
 - Use other synthesis options available in your synthesis tool ([page 6-37](#))
4. Use the DSE Tcl/Tk script as appropriate to automate successive compilations of a design, each employing the different options in steps 1 through 3.
5. Make LogicLock assignments ([page 6-37](#)).

6. Make location assignments, or perform manual placement by back-annotating the design ([page 6-40](#)).

If these options do not achieve performance requirements, design source code modifications may be required ([page 6-45](#)).

Synthesis Netlist Optimizations & Physical Synthesis Optimizations

The Quartus II software offers advanced netlist optimization options, including physical synthesis, for certain device families, to optimize your design further than the optimization performed in the course of the standard Quartus II compilation.

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. 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 the Quartus II integrated synthesis, these optimizations make changes to the synthesis netlist that improve either area or speed, depending on your selected optimization technique.

The following synthesis netlist optimizations are available:

- WYSIWYG primitive resynthesis
- Gate-level register retiming

You can view and modify the synthesis netlist optimization options on the **Synthesis Netlist Optimizations** page under **Analysis & Synthesis Settings** in the **Settings** dialog box (Assignments menu).

The physical synthesis optimizations take place 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:

- Physical synthesis for combinational logic
- Physical synthesis for registers
 - Register duplication
 - Register retiming

You can specify the physical synthesis optimization options on the **Physical Synthesis Optimizations** page under **Fitter Settings** in the **Settings** dialog box (Assignments menu). 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.



For more information and detailed descriptions of these netlist optimization options, see the *Netlist Optimizations & Physical Synthesis* chapter in Volume 2 of the *Quartus II Handbook*.

To achieve the best results, you may need to try these options in different combinations. Performance results are design dependant. You can use the DSE Tcl/Tk script to automate successive compilations of a design, each employing different options. On average, turning on all the options gives the best results. Typical benchmark results with netlists from a leading third-party synthesis tool and compiled with Quartus II software version 4.2 are shown in [Table 6–7](#). These results were obtained for Stratix devices, using various designs and numbers of LEs.

The results for the **WYSIWYG primitive re-synthesis** option depend on the **Optimization Technique** selected on the **Analysis & Synthesis** page of the **Settings** dialog box (Assignments menu). These results use the default **Balanced** setting. Changing the setting to **Speed** or **Area** can affect your results.

Seed

Changing the seed affects the initial placement configuration and often causes different Fitter results. To obtain a better f_{MAX} value, you can experiment with different settings. This method should only be attempted if the design is finalized and is failing timing on a small number of paths. The f_{MAX} variation is typically about 3% for Stratix devices.

Changing the seed changes Fitter results because all fitter algorithms have random variations when initial conditions change, and changing the seed takes advantage of this behavior. However, note that if anything in the design changes, the results change from seed to seed.

The seed for initial placement is controlled by the **Seed** setting on the **Fitter Settings** page of the **Settings** dialog box (Assignments menu).

The DSE Tcl/Tk script can automate successive compilations of a design, each employing different seeds.

Table 6–7. Average Performance of Synthesis Netlist & Physical Synthesis Optimizations for Stratix Designs Notes (1), (2)					
Optimization Method	f _{MAX} Gain (%) (1)	Win Ratio (%) (2)	Winner's f _{MAX} Gain (%) (3)	Logic Area Change (%) (4)	Compile Time Change (x) (5)
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
WYSIWYG primitive resynthesis as well as physical synthesis for combinational logic and registers					
Using physical synthesis Fast effort level	13	87	14	-5	1.4
Using physical synthesis Normal effort level	18	87	19	-5	2.2
Using physical synthesis Extra effort level	20	87	21	-5	3.7
All options on (WYSIWYG primitive resynthesis, gate level register retiming, and physical synthesis for combinational logic and registers)					
Using physical synthesis Extra effort level	20	82	21	-6	3.7

Notes to Table 6–7:

- (1) The average f_{MAX} gain for Stratix II is 10% with a win ratio of 85%.
- (2) The average f_{MAX} gain for Cyclone II is 10% with a win ratio of 80%.
- (3) Win ratio is the percentage of designs that showed better performance with the option on than without the option on.
- (4) Winner's f_{MAX} gain refers to the average improvement for the designs that showed better performance with these settings (designs considered a win).
- (5) Negative values mean reduced area. Positive values mean increased area.
- (6) On average, compile times decreased by 25% from Quartus II version 4.1 to 4.2.



For more information on compiling with different seeds in the DSE script, see the *Design Space Explorer* 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 its performance. Performance varies depending on the way the design is coded, which synthesis tool is used, and which options are specified when synthesizing. Synthesis options should be changed if a large number of paths are failing or specific paths are failing by a large amount and have many levels of logic.

Ensure that you have set your device and timing constraints correctly in your synthesis tool. Your synthesis tool tries to meet the specified requirements. If a target frequency is not specified, some synthesis tools optimize for area.

To achieve best performance with push-button compilation, follow the recommendations in the following paragraphs. You can use the DSE to experiment with different Quartus II synthesis options to optimize for best performance.



For information on setting timing requirements and synthesis options in other synthesis tools, see the appropriate chapter in the *Synthesis* section in Volume 1 of the *Quartus II Handbook*, or see your synthesis software's documentation.

Most synthesis tools optimize to meet your speed requirements. Some synthesis tools offer an easy way to optimize for speed instead of area. For the Quartus II integrated synthesis, specify **Speed** as the **Optimization Technique** option on the **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu). 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).

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 may improve performance. Therefore, whenever possible, flatten your design hierarchy to achieve best results. If you are using the Quartus II integrated synthesis, ensure that the **Preserve Hierarchical Boundary** logic option is turned off (that is, ensure that you have not turned on the option in the Assignment Editor or with Tcl assignments).

Set the Synthesis Effort to High (Where Applicable)

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.

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, choose **One-Hot for State Machine Processing** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu). You can also specify this logic option for specific modules or state machines in your design with the Assignment Editor. In some cases (especially in Stratix II devices), other encoding styles can offer better performance. You can 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 to set the maximum fan-out of a register. In the 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.

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, you can make the entries like the example shown in [Table 6–8](#):

Table 6–8. Entries for the Assignment Editor			
From	To	Assignment Name	Value
My_high_fanout_node	*hierarchy_A*	Manual Logic Duplication	high_fanout_to_A



For more information on the manual logic duplication option, refer to the Quartus II Help.

Use Other Synthesis Options Available in Your Synthesis Tool

With your synthesis tool, experiment with the following options if they are available:

- Register balancing or retiming
- Register pipelining

Note that these options may increase the logic utilization of your design.

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, but re-coding the design is thought to be unnecessary. LogicLock assignments can help if routing delays form a large portion of your critical path delay, and placing logic closer together in the device will help improve the routing delay. This technique is most beneficial for devices with hierarchical routing structures such as the APEX 20K device family.



Note that improving fitting results, especially for larger devices such as Stratix and Stratix II, can be difficult. The LogicLock feature is intended to be used for performance preservation, therefore LogicLock assignments will not always improve the performance of the design. In many cases you will not be able to improve upon the results from the Fitter by making any kind of placement assignments.

When making LogicLock assignments, it is important to consider how much flexibility to leave 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 overconstraint. 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, see the timing analysis results and the Timing Closure Floorplan. The register-to-register f_{MAX} paths in the Timing Analyzer section of the Compilation Report can provide a helpful method of recognizing patterns. The following paragraphs describe cases in which LogicLock regions can help to optimize a design.



For more information on the LogicLock design methodology, see the *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*.

Hierarchy Assignments

For a design with the hierarchy shown in [Figure 6–10](#), which has failing paths in the timing analysis results similar to those shown in [Table 6–9](#), mod_A is probably a problem module. In this case, the mod_A hierarchy block could be placed in a LogicLock region to attempt to put all the nodes in the module closer together in the floorplan.

Figure 6–10. Design Hierarchy

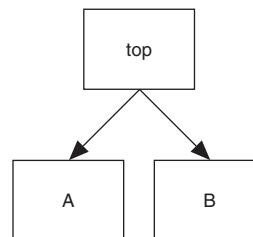


Table 6–9 shows the failing module paths in timing analysis.

Table 6–9. Failing Module Paths 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

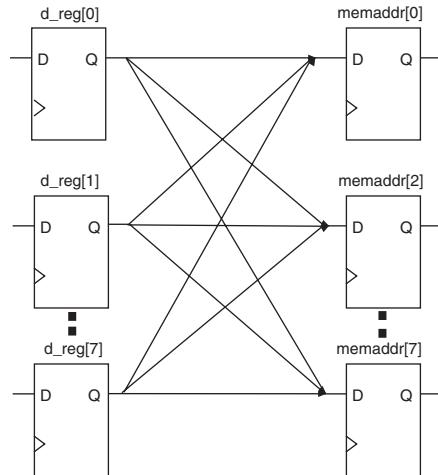
Path Assignments

If you see a pattern such as the one shown in Figure 6–11 and Table 6–10, it is probably an indication of paths with a common problem. In this case, a path-based assignment could be made from all d_reg registers to all memaddr registers. A path-based assignment can be made to place all source registers, destination registers, and the nodes between them in a LogicLock region using the wild cards characters “*” and “?”.

You can also explicitly place the nodes of a critical path in a LogicLock region. There may be alternate paths between the source and destination registers that could become critical if you use this method instead of path-based assignments.



For information on making path-based assignments, using wild cards, and individual node assignments, see the *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*.

Figure 6–11. Failing Paths in Timing Analysis**Table 6–10. 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]

Location Assignments & Back-Annotation

If a small number of paths are failing, 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 may be able to enter location constraints in a way that produces better results.



Note that improving fitting results, especially for larger devices such as Stratix and Stratix II, can be difficult. Location assignments will not always improve the performance of the design. In many cases you will not be able to improve upon the results from the Fitter by making any kind of placement 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. These assigned nodes are then constrained in the region's boundaries. If any portion of a block in the device floorplan, such as an M-RAM block, overlaps with 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

Fixing the location of nodes in a design in the locations resulting from the last compilation is known as back-annotation. When all the nodes are back-annotated, manually moving nodes does not affect the locations of other design nodes that are locked down. This is referred to as manual placement.



Locking down node locations is very restrictive to the Compiler, so you should only back-annotate when the design has been finalized and no further changes are expected. The assignments may 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, and in many cases you will not be able to improve upon the results from the Fitter.

Illegal or unroutable location constraints may cause “no fit” errors.

Before making location assignments, determine whether to lock down the location of all nodes in the design. When you are using a hierarchical design flow, you can choose to lock down node locations in only one LogicLock region, while the other node locations are left as floating in a

fixed LogicLock region. A hierarchical approach using the LogicLock design methodology can reduce the dependence of logic blocks upon other logic blocks in the device.

To perform back-annotation on any portion of a design, that portion of the design must be finalized with no expected changes. Back-annotation uses the node names of the design, so the node names must be constant to allow 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 to fix your node names before locking down the placement of any node.

 Since physical synthesis optimizations are placement-specific as well as design-specific, you may not achieve the same physical synthesis results using an atom netlist generated by the Quartus II software when you recompile if you do not back-annotate the design. This occurs because the different design source (atom netlist instead of original source files) may lead to different placement results. When using an atom netlist, if you need to maintain the same placement results, use LogicLock regions to back-annotate the placement of all nodes in the design.

 For more information on a block-based design approach and creating atom netlists for your design, see the *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*.

When you back-annotate a design, you can choose that the nodes be assigned either to LABs (this is preferred because of increased flexibility) or LEs/ALMs. You can also 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 may cause a routing failure.

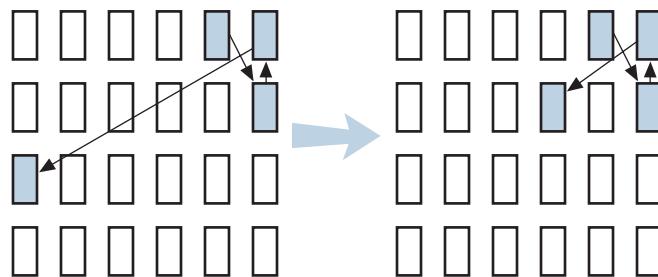
 For more information on back-annotation of routing, see Quartus II Help.

When performing manual placement on a detailed level, Altera suggests 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 place-and-route is more difficult when you move individual logic cells because if LEs with different control signals are put into the same LAB, the LAB may not have any unused control signals available and the design may not fit.

In general, when you are performing manual place-and-route, it is best to fix all I/O paths first. This is because there are often fewer options available to meet I/O timing. After I/O timing has been met, focus on manually placing f_{MAX} paths. This strategy follows 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 6-12](#), moving the destination node closer to the other nodes reduces the delay and may cause it to meet your timing requirements.

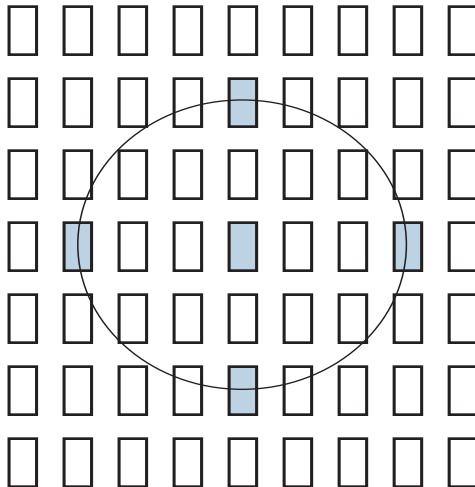
Figure 6-12. Reducing Delay of Critical Path



Optimizing Placement for Stratix II, Stratix, Stratix GX & Cyclone II Devices

In Stratix II, Stratix, Stratix GX, and Cyclone II architectures, 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 6-13](#), if the source is located in the center, any of the shaded LABs should give approximately the same delay.

Figure 6–13. Possible Optimal Placement Ellipse



In addition, you should avoid crossing any M-RAM memory blocks for node-to-node routing, if possible, because routing paths across M-RAM blocks requires using *R24* or *C16* routing lines.

To determine the actual delays to and from a resource, use the **Show Physical Timing Estimate** feature in the Timing Closure Floorplan.



For more information on using the Timing Closure Floorplan, see the *Timing Closure 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 because they require more than one routing line to cross the Cyclone device.

Optimizing Placement for Mercury, APEX II & APEX 20KE/C Devices

For the Mercury, APEX II, and APEX 20KE/C architectures, the delay for paths should be reduced by placing the source and destination nodes in the same geographical resource location. The following list shows the device resources 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, they should be placed 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.

Optimize Source Code

If the methods described in the preceding sections do not sufficiently improve the timing in the design, you must modify the design at the source to achieve the desired results. You may be able to re-architect the design using 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 often a better choice than using LogicLock or location assignments.

If your critical path involves memory or DSP functions, check whether 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 cause these functions to be placed into high-performance dedicated memory or DSP resources in the target device.



For coding style guidelines including examples of HDL code for inferring memory and DSP functions, refer to the *Inferring and Instantiating Altera Megafunctions* section of the *Recommended HDL Coding Styles* chapter in Volume 1 of the *Quartus II Handbook*.

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** (Processing menu). 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 guidelines and sample HDL code for state machines, refer to the *State Machines* section in the *Recommended HDL Coding Styles* chapter in Volume 1 of the *Quartus II Handbook*.

Resource Utilization Optimization Techniques (Macrocell-Based CPLDs)

The following recommendations will 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](#)” on page 6–3. 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 can also 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 disadvantages to using logic-generated controls signals include:

- More resources are required (logic cells, interconnect)
- May result in more data skew
- 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 several ways:

- In the Assignment Editor, choose one of two methods:
 - Assign pins to dedicated pin locations
 - Assign a **Global Signal** setting to the pins
- Choose **Register Control Signals** in the **Auto Global Options** section of the **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu)
- Insert a **GLOBAL** primitive after the pins



If you have already assigned pins for the design in the MAX+PLUS® II software, choose **Import Assignments** (Assignments menu).

Reserve Device Resources

Because pin and logic option assignments might be necessary for board layout and performance requirements, and because full utilization of the device resources may cause the design to be more difficult to fit, 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 increases the chance that the Quartus II software will be able to fit the design during re-compilation after changes or assignments have been made.

Pin Assignment Guidelines & 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 appropriate LAB
- Assign output pins in need of 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

You should 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, GCLRn). You can assign clocks to global clock dedicated inputs (GCLK1, OE2/GCLK2), clear to the global clear dedicated input (GCLRn), and speed-critical output enable to global OE dedicated inputs (OE1, OE2/GCLK2).

Figure 6–14 shows the EPM3032A device’s pin-out information for the dedicated pins. You can use the Quartus II Help to determine the dedicated input pin numbers.

Figure 6–14. EPM3032A Dedicated Pin-Out Information Shown in Quartus II Help

Function	Dedicated Input Pins				Pin Numbers		
	Config. Pin Note (10)	LCell	OE MUX Pin:LCell	LAB	IO Bank	PLCC 44	TQFP 44
Input/GCLK	-	-	-/-	-	-	43	37
Input/OE1n	-	-	1/-	-	-	44	38
Input/GCLRn	-	-	-/-	-	-	1	39
Input/OE2n/GCLK	-	-	2/-	-	-	2	40

Output Enable Pin Assignments

Occasionally, because the total number of required output enable pins is more than the dedicated input pins, output enable signals may need to be assigned to I/O pins. Therefore, to minimize the possibility of fitting errors, refer to Quartus II Help when assigning the output enable pins for MAX 7000 and MAX 3000 devices. Search for the device name (for example, EPM3032A) in Quartus II Help to locate the device pin table.

Figure 6–15 shows the dedicated pin-out information for the EPM3512A device from Quartus II Help. Specifically, Figure 6–15 shows that the first row *Pin;LCell* value is 8/5; which means that GOE8 can be driven by pin 170 or C6 (depending on package) and GOE5 can be driven by logic cell 21.

Figure 6–15. EPM3512A Dedicated Pin-Out Information in Quartus II Help

This column lists the possible sources for
the output enable signals (i.e., GOE1, GOE2, etc), (1)

Function	Config. Pin <u>Note (10)</u>	LCell	OE MUX Pin; LCell	LAB	IO Bank	TQFP 144	PQFP 208
I/O or Buried	-	21	8<5>	B	-	170	C6
I/O or Buried	-	22	-/-	B	-	-	-
I/O or Buried	-	23	-/4	B	-	-	-
I/O or Buried	-	24	-/9	B	-	-	-
I/O or Buried	-	25	(3)>2	B	-	171	B6
I/O or Buried	-	26	-/-	B	-	-	-
I/O or Buried	-	27	4/1	B	-	172	A6
I/O or Buried	-	28	-/7	B	-	-	-
I/O or Buried	-	29	-/10	B	-	-	-
I/O or Buried	-	30	-/-	B	-	-	F7

Global output enable signals that are fed by
pin in the corresponding Function column.

Global output enable signals that are fed by
logic cell in the corresponding LCell column.

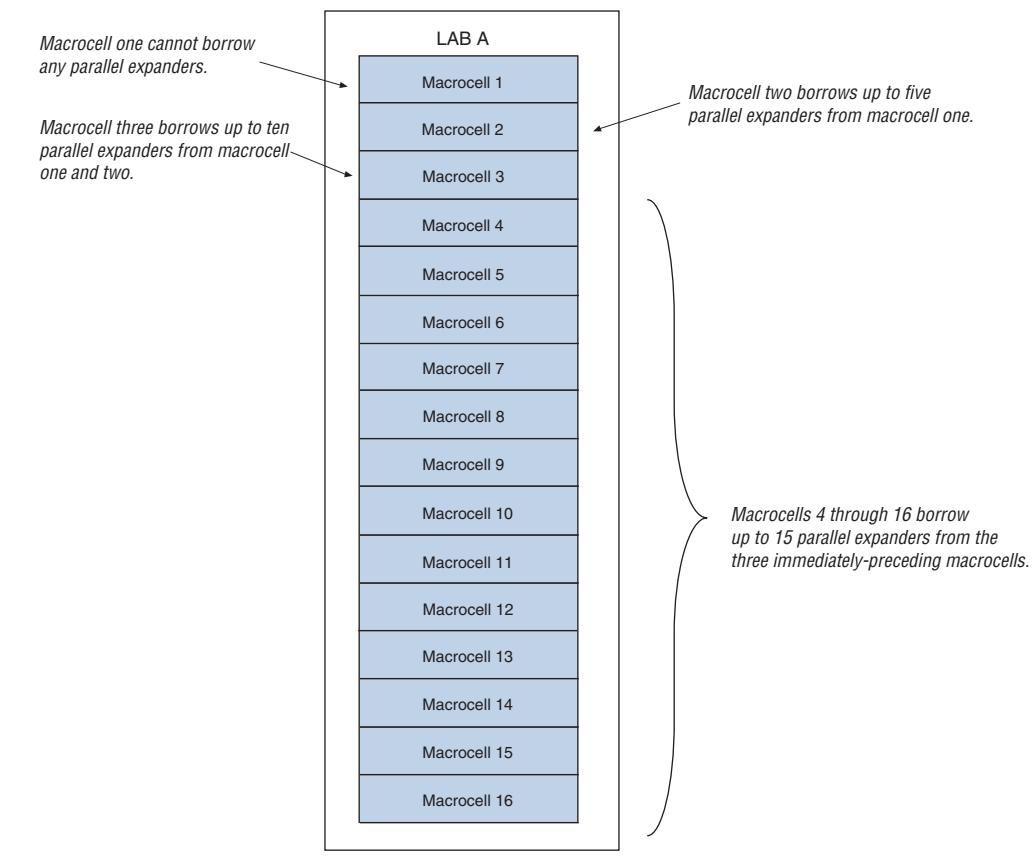
Estimate Fan-In When Assigning Output Pins

Macrocells with high fan-in can cause more placement problems for the Quartus II Fitter than those with low fan-in. The maximum number of fan-in per LAB should not exceed 36 in MAX 7000 and MAX 3000 devices. Therefore, it is important to 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 6–16 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 6–16**, only macrocells 2 through 16 can borrow parallel expanders. Therefore, you should 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 6–16. LAB Macrocells & Parallel Expander Associations

Resolving Resource Utilization Problems

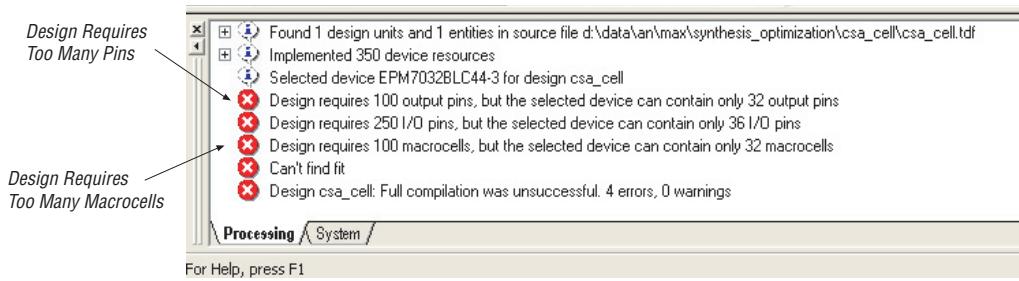
During compilation with the Quartus II software, you may receive an error message (see [Figure 6–17](#)) alerting you that the compilation was not successful.

There are two common Quartus II compilation fitting issues: macrocell usage and 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 cannot implement the design. To resolve your design issues, check the Message Window (see [Figure 6–17](#)) for the no-fit compilation results.



Messages in the Message Window are also copied in the Report Files. Right-click on a message and select **Help** (right button pop-up menu) for more information.

Figure 6–17. Quartus II Software Compilation No-Fit Error Message Window



Resolving Macrocell Usage Issues

Occasionally, a design requires more macrocell resources than are available in the selected device, resulting in a no-fit compilation. The following list provides tips for resolving macrocell-usage issues as well as tips to minimize the amount of macrocells used:

- Turn off **Auto Parallel Expanders** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu)—If the design's clock frequency (f_{MAX}) is not an important part of the design requirements, you should turn off the parallel expanders for all or part of the project. The design will usually require 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}). You can change the **Optimization Technique** option in the **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu).
- Use D-flip-flops instead of latches—Altera recommends that you always use D-flip-flops instead of latches in your design because D-flip-flops may 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 individual MAX 7000 and MAX 3000 macrocells contain D-flip-flops 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 will produce macrocells and pins with higher fan-out.



If you have followed the suggestions listed in this section and your project still does not fit the targeted device, consider using a larger device. When upgrading to a different density device, the vertical-package-migration feature of the MAX 7000 and MAX 3000 device families allows pin assignments to be maintained.

Resolving Routing Issues

The other resource that can cause design-fitting issues is routing. For example, if the total fan-in into a LAB exceeds the maximum allowed, the result may be a no-fit error 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. Therefore, Altera recommends that you 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 the macrocell usage issues. See the same suggestion in “[Resolving Macrocell Usage Issues](#)” on page 6–51.
- 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% and 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** (Assignments menu) or under **More Settings** in the **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu).
- Turn off **Auto Parallel Expanders** in the **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu)—By turning off the parallel expanders, the Quartus II software has more fitting flexibility for each macrocell, allowing macrocells to be relocated. For example, each macrocell (previously grouped together in the same LAB) may be moved to a different LAB to reduce routing constraints.

- Inserting logic cells—Inserting logic cells reduces fan-in and shared expanders used per macrocell, increasing routability. By default, the Quartus II software will automatically insert logic cells when necessary. You can turn this feature off by turning off **Auto Logic Cell Insertion** under **More Settings** in the **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu). See “[Using LCELL Buffers to Reduce Required Resources](#)” on page 6–53 for more information.
- Change pin assignments—if you are willing to discard your pin assignments, you can let the Quartus II Fitter automatically ignore all the assignments, the minimum number of assignments, or specific assignments.



If you prefer reassigning the pins to increase the device routing efficiency, refer to “[Pin Assignment Guidelines & Procedures](#)” on page 6–47.

Using LCELL Buffers to Reduce Required Resources

Complex logic, such as multi-level 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 can also break down complex logic by inserting logic cells in the project to reduce the average fan-in and total number of shareable expanders needed. Manually inserting logic cells can provide greater control over speed-critical paths.

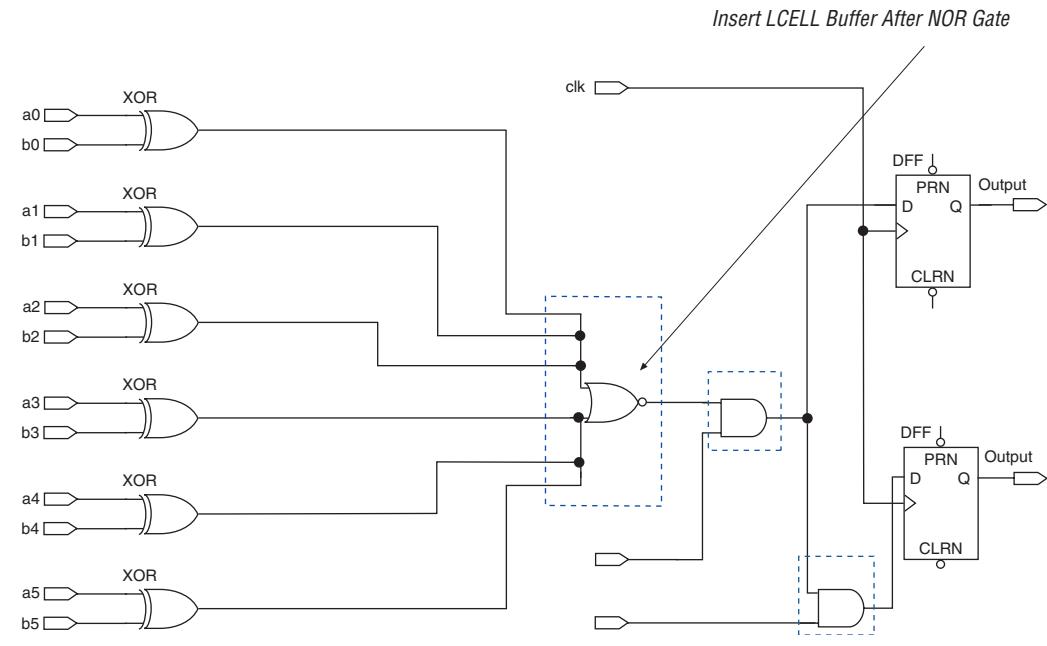
Instead of using the Quartus II software’s **Auto Logic Cell Insertion** option, you can manually insert logic cells. However, Altera recommends that you use the **Auto Logic Cell Insertion** option unless you know which part of the design is causing the congestion.

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 duplicating 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, see [Figure 6–18](#). The [Figure 6–18](#) design was compiled for a MAX 7000AE device. Without the LCELL buffer, the design requires two macrocells, eight shareable

expanders, and the average fan-in is 14.5. However, with the LCELL buffer, the design requires three macrocells, eight shareable expanders, and the average fan-in is just 6.33.

Figure 6–18. Reducing the Average Fan-In by Inserting LCELL Buffers



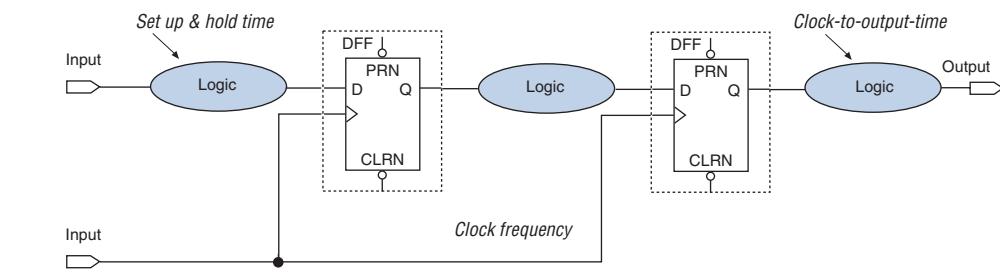
Timing Optimization Techniques (Macrocell-Based CPLDs)

The stage of design optimization after resource optimization focuses on timing. Ensure that you have made the appropriate assignments as described in “Initial Compilation” on page 6–3, and that the resource utilization is satisfactory, before proceeding with timing optimization.

Maintaining the system’s performance at or above certain timing requirements is an important goal of circuit designs. The five main timing parameters that determine a design’s system performance are: setup time (t_{SU}), hold time (t_H), clock-to-output time (t_{CO}), pin-to-pin delays (t_{PD}), and maximum clock frequency (f_{MAX}). The setup and hold times are the propagation time for input data signals. Clock-to-output time is the propagation time for output signals, pin-to-pin delay is the time required for a signal from an input pin to propagate through combinational logic and appear at an external output pin, and the maximum clock frequency is the internal register-to-register performance.

This section provides guidelines to improve the timing if the timing requirements are not met. [Figure 6–19](#) shows the parts of the design that determine the t_{SU} , t_H , t_{CO} , t_{PD} , and f_{MAX} timing parameters.

Figure 6–19. 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, as discussed in “[Design Analysis](#)” on page 6–8.

When you are analyzing a design to improve its performance, be sure to consider the two major contributors to long delay paths:

- Excessive levels of logic
- Excessive loading (high fan-out)

For MAX 7000 and MAX 3000 devices, when a 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, you should 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 may 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 helpful when a pin drives a D-flip-flop without 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 will cause more delays. Therefore, 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. Therefore, to improve the setup time, minimize the fan-out.

Improving Clock-to-Output Time

To improve a design's clock-to-output time, you should minimize the register-to-output-pin delay. To improve the t_{CO} timing, refer to the following guidelines:

- Use the global clock—Besides minimizing the delay from the register to output pin, minimizing the delay from the clock pin to the register can also improve the t_{CO} timing. Altera recommends that you 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 will cause more delay. Always minimize the amount of logic between the register and output pin for faster clock-to-output time.

Table 6-11 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} timing will increase.

Table 6–11. 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	No	On	LAB A	LAB A	No	No
1	1.6	0.3	4.3	No	Off	LAB A	LAB A	No	No
1	2.5	0	3.1	Yes	On	LAB A	LAB A	No	No
1	2.8	0	3.1	Yes	Off	LAB A	LAB A	No	No
1	3.6	0	3.1	Yes	Off	LAB A	LAB A	Yes	No
1	2.8	0	7.0	Yes	Off	LAB D	LAB A	No	Yes
16 with the same D and clock inputs	2.8	0	All 6.2	Yes	Off	LAB D	LAB A, B	No	No
32 with the same D and clock inputs	2.8	0	All 6.4	Yes	Off	LAB C	LAB A, B, C	No	No

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} , Altera recommends that you follow the guidelines discussed in this section.

- Turn on Auto Parallel Expanders in the Analysis & Synthesis Settings page of the **Settings** dialog box (Assignments menu)—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 will also increase and result in a no fit error during compilation if the device's resources are limited. For more information about turning the **Auto Parallel Expanders** option on, refer to “[Resolving Macrocell Usage Issues](#)” on page 6–51.

- 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. Thus, you should only need to reset the **Optimization Technique** option back to **Speed** if you have previously set it to **Area**. You can reset the **Optimization Technique** option in the **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu).

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 these guidelines to improve your design's clock speed (f_{MAX}).

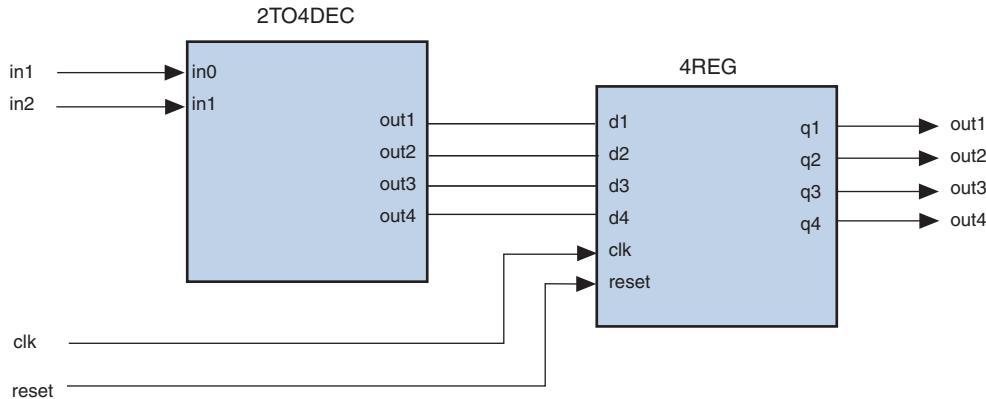
- Turn on **Auto Parallel Expanders** in the **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu)—Turning on the parallel expanders for individual nodes or sub-designs can increase the performance of complex logic functions. However, if the project's pin or logic cell assignments use parallel expanders placed physically together with macrocells (which can reduce routability), parallel expanders can cause the Quartus II Compiler to have difficulties finding and optimizing a fit. Additionally, the amount of macrocells required to implement the design will also increase and result in a no fit error during compilation if the device's resources are limited. For more information on turning the **Auto Parallel Expanders** option on, refer to [“Resolving Macrocell Usage Issues” on page 6–51](#).
- Use global signals/dedicated inputs—Altera MAX 7000 and MAX 3000 devices' dedicated inputs provide low skew and high speed for high fan-out signals. Thus, Altera recommends that you always 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. Thus, you should need to 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 **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu).
- Pipeline the design—Pipelining, which increases clock frequency (f_{MAX}), refers to dividing large blocks of combinational logic by inserting registers. For more information on pipelining, see [“Optimizing Source Code—Pipelining for Complex Register Logic”](#).

Optimizing Source Code—Pipelining for Complex Register Logic

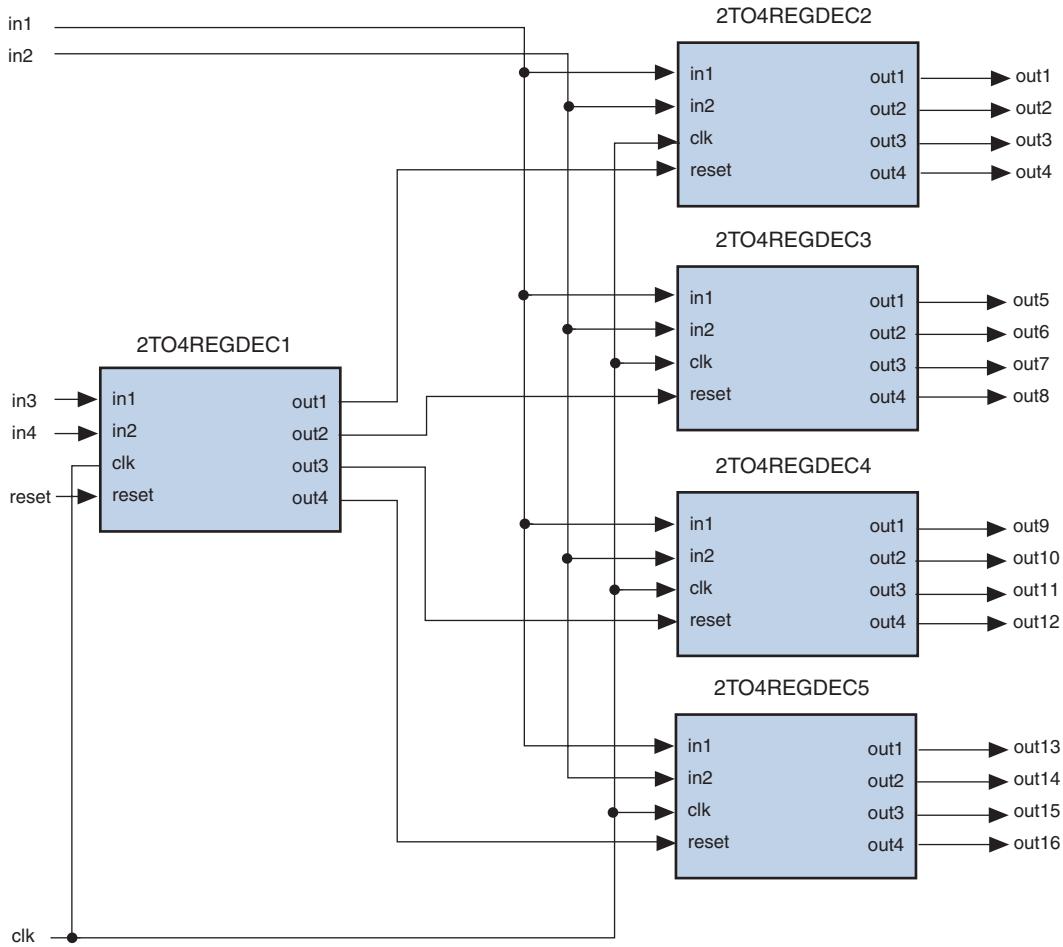
If the methods described in the preceding sections do not sufficiently improve your results, modify the design at the source to achieve the desired results. Using a pipelining technique can consume device resources, but it also lowers the propagation delay between registers, allowing you to maintain 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-flip-flops. [Figure 6–20](#) 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-flip-flops in 4REG.

Figure 6–20. A 2- to 4-Pipelined Decoder



[Figure 6–21](#) shows five 2-to-4 decoders (2TO4REGDEC) that are combined to form a 4-to-16 pipelined decoder. The first decoder (2TO4REGDEC1) will decode the two most significant bits (MSB) (in_3 and in_4) of the 4-to-16 decoder. The decoded output from the 2TO4REGDEC1 decoder will only enable one of the rest of the 2-to-4 decoders (2TO4REGDEC2, 2TO4REGDEC3, 2TO4REGDEC4, or 2TO4REGDEC5). The inputs in_1 and in_2 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 6–21](#), the MSBs (in_3 and in_4) are decoded in the first clock cycle, while the other bits (in_1 and in_2) are decoded in the following clock cycle.

Figure 6–21. Five 2-to-4 Pipelined Decoders Combined to Form a 4-to-16 Pipelined Decoder

Compilation Time Optimization Techniques

If optimizing 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.

Reducing Synthesis & Synthesis Netlist Optimization Time

The time spent doing synthesis can be reduced by reducing your use of synthesis netlist optimizations and by using incremental synthesis. For more ideas on reducing synthesis time in third-party EDA synthesis tools, refer to your tool'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 make the Analysis & Synthesis module take much longer to run. Look at the Analysis & Synthesis messages to find out how much time these optimizations take. Note that the compilation time spent in Analysis & 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 the optimizations off 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.

Incremental Synthesis

In the Quartus II software, you can use the incremental synthesis feature to save synthesis time when making changes to the design. Incremental synthesis allows you to set design partitions to ensure that only those sections of a design that have been updated will be resynthesized when the design is compiled, reducing synthesis time and run-time memory usage.

If using a third-party synthesis tool, you can create separate atom netlists for parts of your design you have already synthesized and optimized so that you only need to update the parts of the design that change.



For information on Quartus II incremental synthesis, refer to the *Quartus II Integrated Synthesis* chapter in Volume 1 of the *Quartus II Handbook*. For information on creating multiple netlist files in third-party tools, refer to the appropriate chapter in the *Synthesis* section in Volume 1 of the *Quartus II Handbook*.

Check Early Timing Estimation Before Fitting

The Quartus II software allows you to get 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 6–5.

Choose **Start > Start Early Timing Estimate** (Processing menu) after you have performed analysis and synthesis in the Quartus II software.

Reducing 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
- 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, or use LogicLock regions to manually control where parts of the design are placed. 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 cancel out the time reduction.

Fitter Effort Setting

Use the **Fitter effort** setting on the **Fitter Settings** page of the **Settings** dialog box (Assignments menu) to shorten run time by changing the effort level to **Auto Fit** or **Fast Fit**.

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 6–7 on page 6–34](#) for detailed results.

If your design meets your performance requirements without physical synthesis options, turn them off to save time. You can also use the **Physical synthesis effort** setting on the **Physical Synthesis Optimizations** page under **Fitter Settings** in the **Settings** dialog box (Assignments menu) to reduce the amount of extra compilation time used by these optimizations. The **Fast** setting directs the Quartus II software to use a lower level of physical synthesis optimization that, compared to the normal level, may cause a smaller increase in compilation time. However, the lower level of optimization may result in a smaller increase in design performance.

Preserving Placement with LogicLock Regions

Preserving information about previous placements can make future placements take less time. To successfully preserve information, node names must not change from placement to placement, and node locations must be preserved so they will not change from placement to placement.

To preserve node names, you must use an atom netlist. Atom netlists include Verilog Quartus Mapping (.vqm) files and EDIF files, which are the outputs of third-party synthesis software. If you use Quartus II integrated synthesis, or turn on any Quartus II netlist optimizations, you must generate a VQM file and turn off netlist optimizations in future compilations.

To preserve node locations, use back-annotated LogicLock regions. After you back-annotate a LogicLock region, the node locations are fixed and the placer skips those nodes, saving time. If you change part of your design in a back-annotated LogicLock region, delete the back-annotated contents of the region and recompile the design. The placer will find a new placement for the changed logic and any logic that is not in a LogicLock region.

Follow these steps to reduce placement time with atom netlists and LogicLock regions:

1. Choose hierarchies in your design to assign to LogicLock regions. You do not have to use LogicLock regions for all hierarchies in your design, just the hierarchies for which you want to reduce placement time.
2. Create separate atom netlists for the chosen hierarchies and assign them to LogicLock regions.
3. Turn off netlist optimizations on each LogicLock region.
4. Compile the design.
5. Back-annotate the LogicLock regions.

Follow these steps when you change logic in a back-annotated LogicLock region:

1. Create a new atom netlist for the hierarchy.
2. Delete the back-annotated contents of the appropriate LogicLock region.
3. Recompile the design.
4. Back-annotate the LogicLock region.



For more information on creating hierarchical designs with multiple netlists, refer to the *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*.

Reducing 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 LogicLock regions to preserve routing information

Routing Congestion

To identify congested routing areas in your design, open the Timing Closure Floorplan. Choose **Timing Closure Floorplan** (Assignments menu) and turn on **Show Routing Congestion**. (Note that this feature is available only when you have chosen to use the **Field View** (View menu)). A routing resource usage above 90% indicates routing congestion.

If the area with routing congestion is in a LogicLock region or between LogicLock regions, remove the LogicLock regions and recompile the design. If the routing time remains the same, then the time is a characteristic of the design and the placement. If the routing time decreases, you should consider changing the size, location, or contents of the LogicLock regions to reduce congestion and decrease routing time.

LogicLock Regions

You can use LogicLock regions back-annotated to the routing level to preserve routing information between compilations. This can reduce the time required to route a design. Follow the same steps as for using LogicLock regions to reduce placement time, but back-annotate to the routing level.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. 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 `quartus_sh --qhelp` at a system command prompt to run the Quartus II Command-Line and Tcl API Help browser.



For more information on Quartus II scripting support, including examples, refer to the *Tcl Scripting* and *Command-Line Scripting* chapters 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>
```

Initial Compilation Settings

Table 6–12. Initial Compilation Settings

Table 6–12 lists the QSF variable name and applicable values for the settings discussed in “Initial Compilation” on page 6–3. 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.

Setting Name	QSF Variable Name	Values	Type
Device Setting	DEVICE	<device part number>	Global
Use Smart Compilation	SPEED_DISK_USAGE_TRADEOFF	SMART, NORMAL	Global
Optimize Timing	OPTIMIZE_TIMING	OFF, "NORMAL COMPILE", "EXTRA EFFORT"	Global
Optimize I/O Cell Register Placement	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

Resource Utilization Optimization Techniques (LUT-Based Devices)

Table 6–13 lists the QSF variable name and applicable values for the settings discussed in “Resource Utilization Optimization Techniques (LUT-Based Devices)” on page 6–14. 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 6–13. Resource Utilization Optimization Settings			
Setting Name	QSF Variable Name	Values	Type
Auto Packed Registers	AUTO_PACKED_REGISTERS_<Device Family Name>	OFF, NORMAL, "MINIMIZE AREA"	Global, Instance
Auto Packed Registers	AUTO_PACKED_REGISTERS_<CYCLONE MAXII STRATIX STRATIXII>	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
State Machine Encoding	STATE_MACHINE_PROCESSING	AUTO, "ONE-HOT", "MINIMAL BITS", "USER-ENCODED"	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 DSP Block Replacement	AUTO_DSP_RECOGNITION	ON, OFF	Global, Instance

I/O Timing Optimization Techniques (LUT-Based Devices)

Table 6–14 lists the QSF variable name and applicable values for the settings discussed in “I/O Timing Optimization Techniques (LUT-Based Devices)” on page 6–24. 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 6–14. I/O Timing Optimization Settings

Setting Name	QSF Variable Name	Values	Type
Optimize I/O cell 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

f_{MAX} Timing Optimization Techniques (LUT-Based Devices)

Table 6–15 lists the QSF variable name and applicable values for the settings discussed in “f_{MAX} Timing Optimization Techniques (LUT-Based Devices)” on page 6–31. 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 6–15. F_{MAX} Timing Optimization Settings

Setting Name	QSF 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
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
Seed	SEED	<integer>	Global
Maximum Fan-Out	MAX_FANOUT	<integer>	Instance
Manual Logic Duplication	DUPPLICATE_ATOM	<node name>	Instance

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 such as the following:

```
set_instance_assignment -name DUPLICATE_ATOM \
    high_fanout_to_A -from high_fanout_node \
    -to *hierarchy_A*
```

Conclusion

Today's complex designs have complex requirements. Methodologies for fitting your design and for achieving timing closure are fundamental to optimal performance in today's designs. 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 effectively 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.

qii52006 2.1

Introduction

With FPGA designs surpassing the million-gate mark, designers need advanced tools to better analyze timing closure issues to achieve their system performance goals.

The Altera® Quartus® II software offers many advanced design analysis tools that allow detailed timing analysis of your designs, including a fully integrated Timing Closure Floorplan Editor. With these tools and options, you can easily determine and locate the critical paths in the floorplan of the targeted device. This chapter explains how to use these tools and options to enhance your FPGA design analysis.

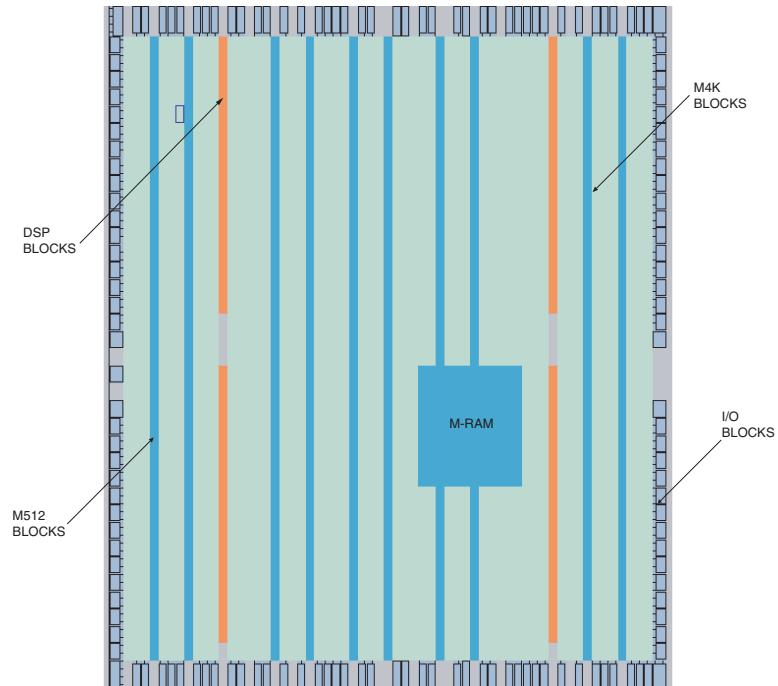
Design Analysis Using the Timing Closure Floorplan

The Timing Closure Floorplan Editor assists you in visually analyzing your designs before and after performing a full design compilation in the Quartus II software. This floorplan editor, used in conjunction with traditional Quartus II timing analysis features, provides a powerful method for performing design analysis.

Timing Closure Floorplan Views

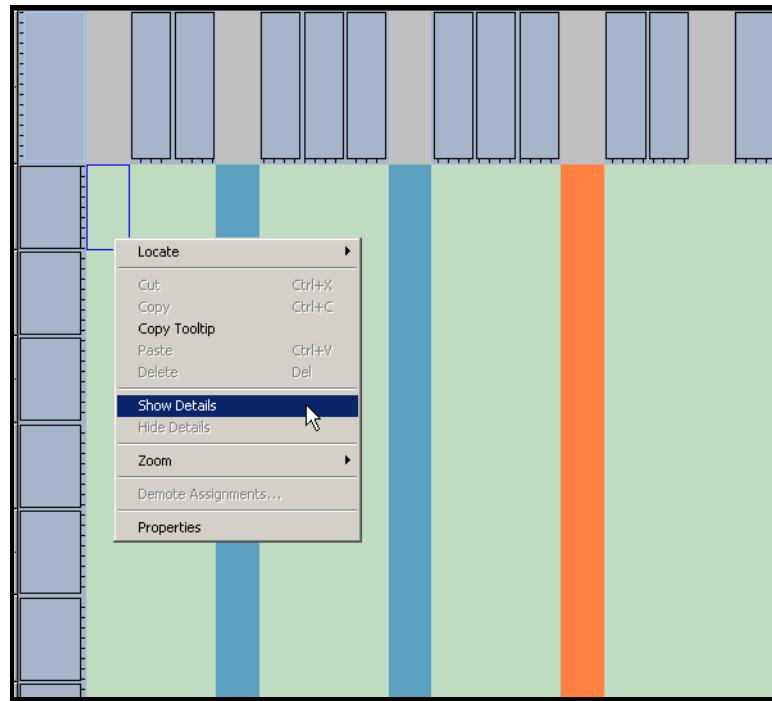
The Timing Closure Floorplan Editor allows you to customize the views of your design. The Field View is a color-coded, high-level view of resources. [Figure 7–1](#) shows the Field View of a Stratix® II device.

Figure 7–1. Field View of a Stratix II Device



In the Field View, you can view the details of a resource by selecting the **resource**, right-clicking, then selecting **Show Details** from the right-button pop-up menu. To hide the details, select all the resources, right-click, and select **Hide Details**. See [Figure 7–2](#).

You can also view your design in the Timing Closure Floorplan Editor with the traditional Interior Cells, Package Top, and Package Bottom views. Use the View menu to change to the various floorplan views.

Figure 7–2. Show Details & Hide Details of a Logic Array Block in Field View

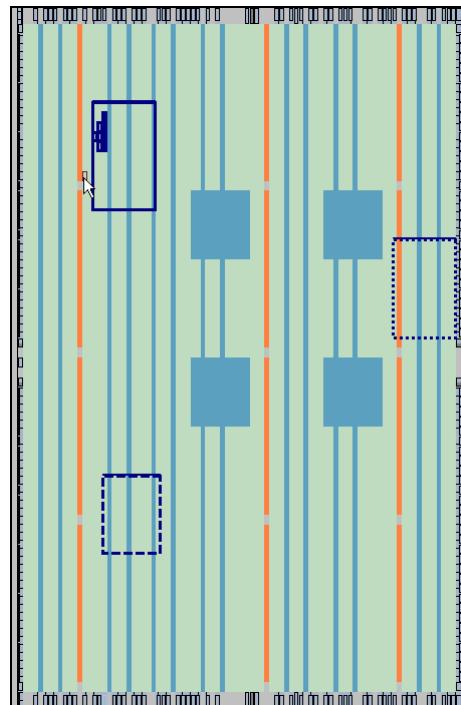
Viewing Assignments

The Timing Closure Floorplan Editor differentiates between user and fitter placements. User assignments are assignments that a user makes including LogicLock™ regions. Fitter placements are the locations where the Quartus II software placed unconstrained [or unassigned] nodes in the last compilation. You can view both user and fitter placements at the same time.

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 newly selected device's floorplan.

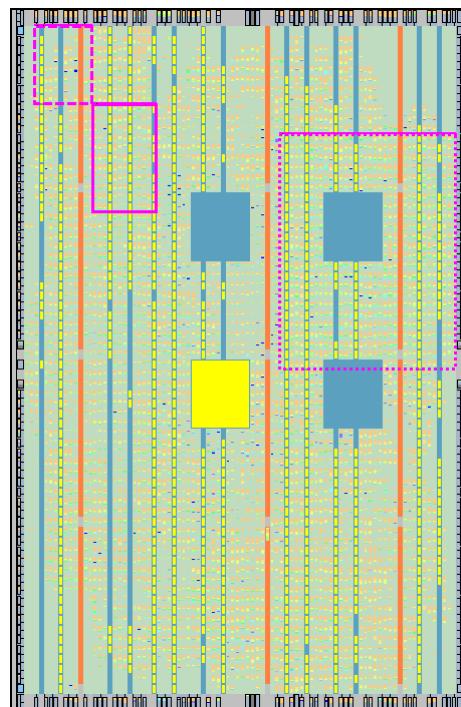
To see user assignments, click the **User Assignments** icon in the **Floorplan Editor** toolbar, or choose **Assignments** (View menu) and select **Show User Assignments**. See [Figure 7–3](#).

Figure 7–3. User Assignments



To see fitter placements, click the **Fitter Placements** icon in the **Floorplan Editor** toolbar, or choose **Assignments** (View menu) and select **Show Fitter Placements**. See Figure 7–4.

Figure 7–4. Fitter Placements

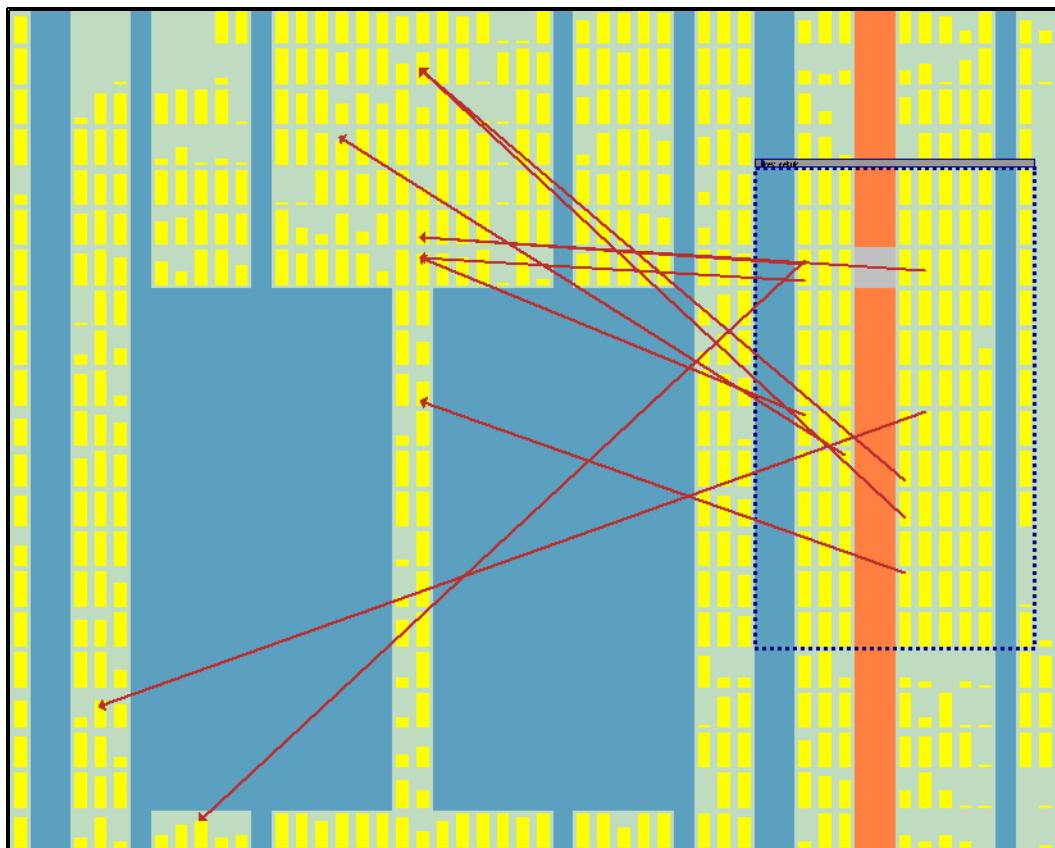


Viewing Critical Paths

The View Critical Paths feature displays routing paths in the Timing Closure floorplan, as shown in [Figure 7–5](#). The criticality of a path is determined by its slack and shown in the timing analysis report.

To view critical paths in the floorplan, click the **Show Critical Paths** icon or chose **Routing > Show Critical Paths** (View menu). To set the criteria for the critical path you want to view, select the **Critical Paths Settings** icon or choose **Routing > Critical Paths Settings** (View menu). See [Figure 7–5](#).

Figure 7–5. Critical Paths

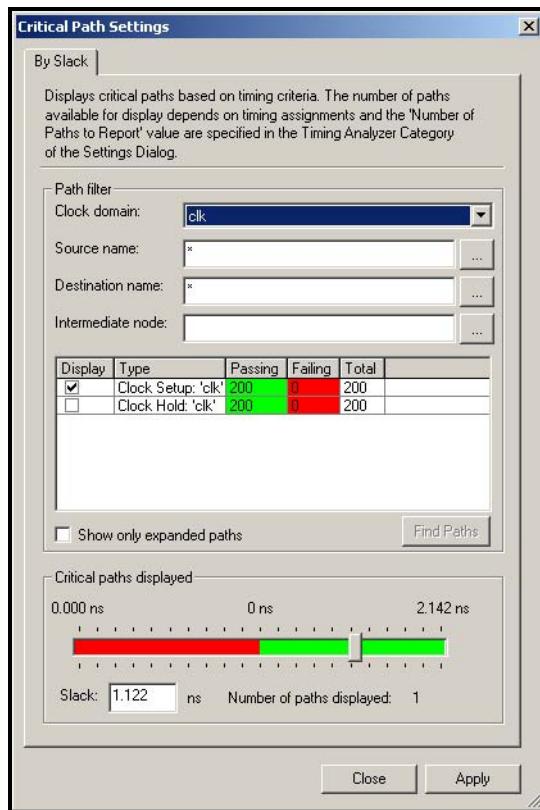


When viewing critical paths, you can specify the clock in the design to be viewed. You determine the paths to be displayed by specifying the slack threshold in the slack field. For example, you can view all paths with a slack of 4.5 ns or worse.



Timing settings must be made and a timing analysis performed for paths to be displayed in the floorplan.

Figure 7–6. Critical Paths Settings Window



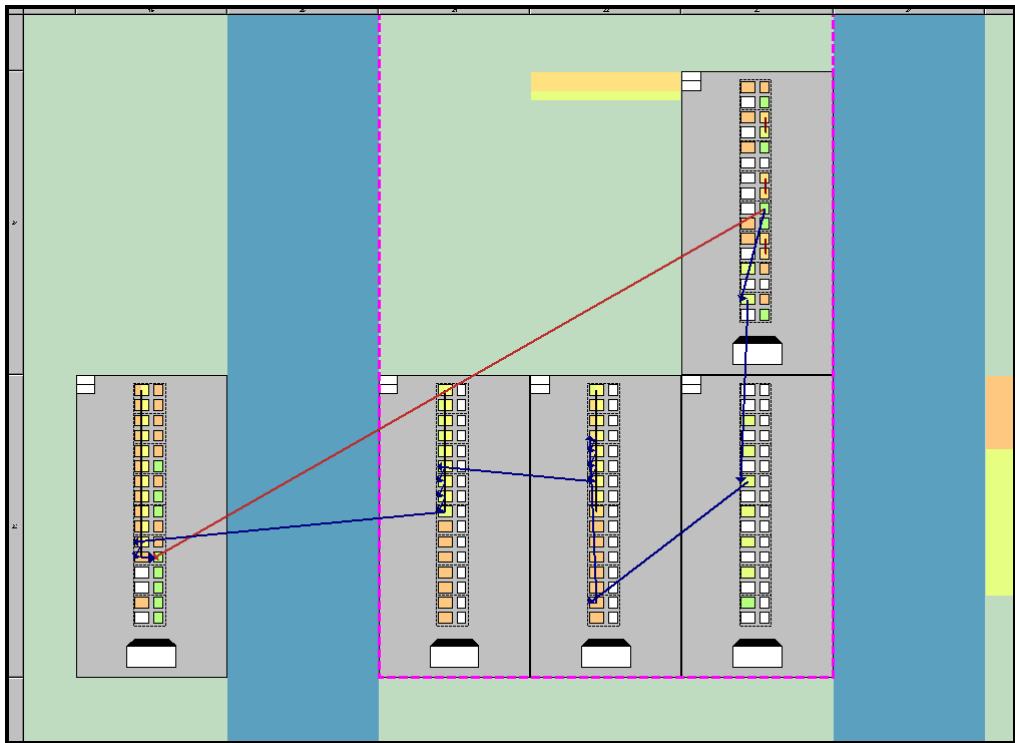
The critical path feature is extremely useful in determining the criticality of nodes based on placement. There are a number of options to view the details of the critical path.

The default view 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, select the path by clicking on the delay label, right click, and select **Show Path Edges**. Figure 7–7 shows a 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.

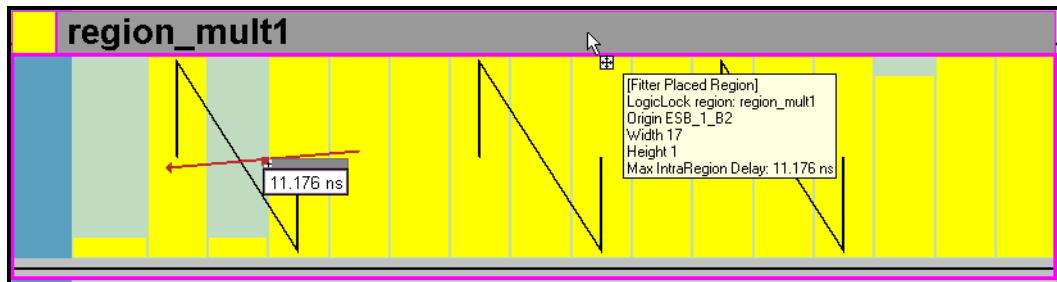
Figure 7–7. Worst-Case Combinational Paths of Critical Paths



You can also assign the path to a LogicLock region in 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, click the **Show Intra-region Delay** icon or go to **Routing > Show Intra-region Delay** (View menu). Place your cursor over a filter-placed LogicLock region to see the maximum delay. [Figure 7–8](#) shows the maximum routing delay of a LogicLock region.

Figure 7–8. Maximum Intra-Region Delay

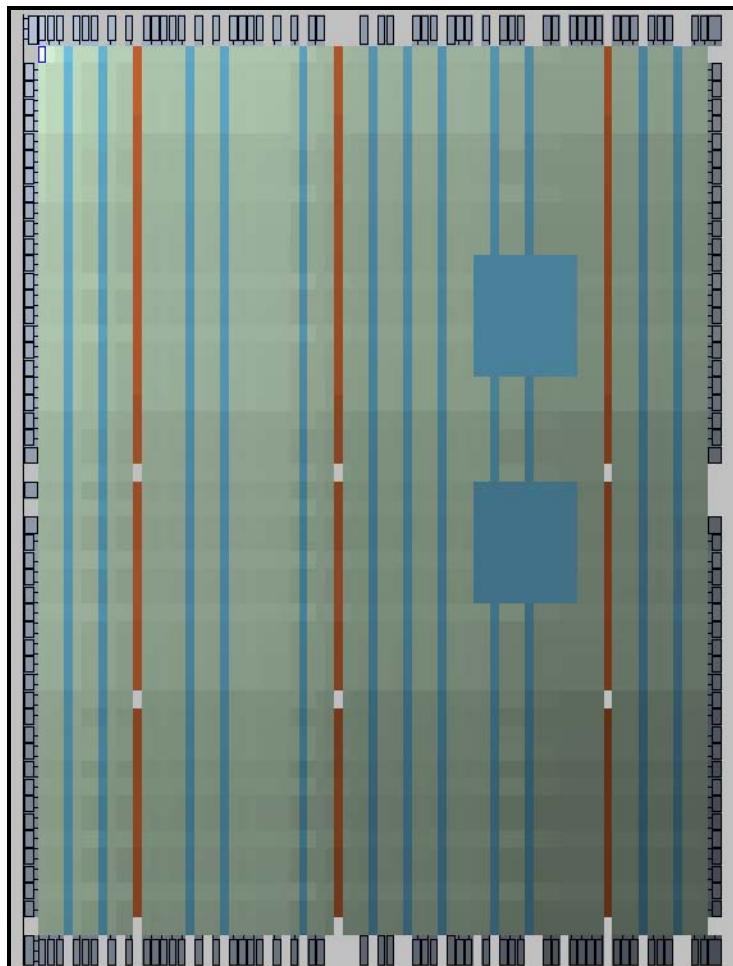


For more information on making path assignments with the **Paths** dialog box, see the *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*.

Physical Timing Estimates

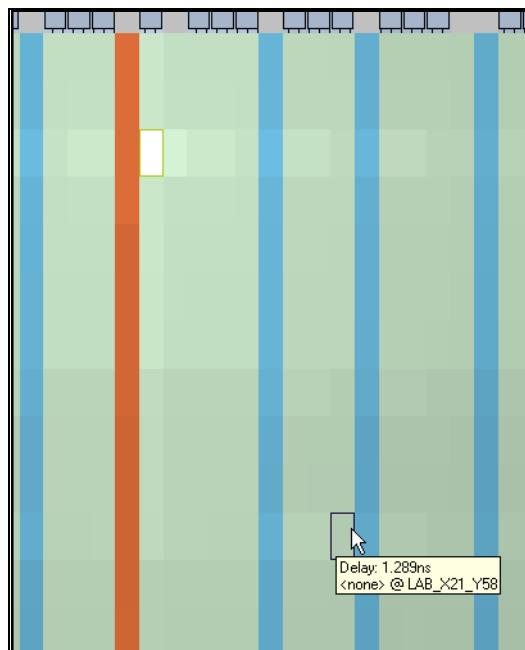
In the Timing Closure Floorplan Editor, you can select a resource and see the approximate delay to any other resource on the device. Once you select a resource, the delay is represented by the color of potential destination resources. The darker the color of the resource, the longer the delay, as shown in [Figure 7–9](#).

Figure 7–9. Physical Timing Estimates View



You can also get an approximation of the delay between two points by selecting a source and holding your cursor over a potential destination resource, as Figure 7-10 shows.

Figure 7-10. Delay for Physical Timing Estimate



The delays represent an estimate based on probable best-case routing. It is possible the delay is 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 choose **Routing > Show Physical Timing Estimates** (View menu).

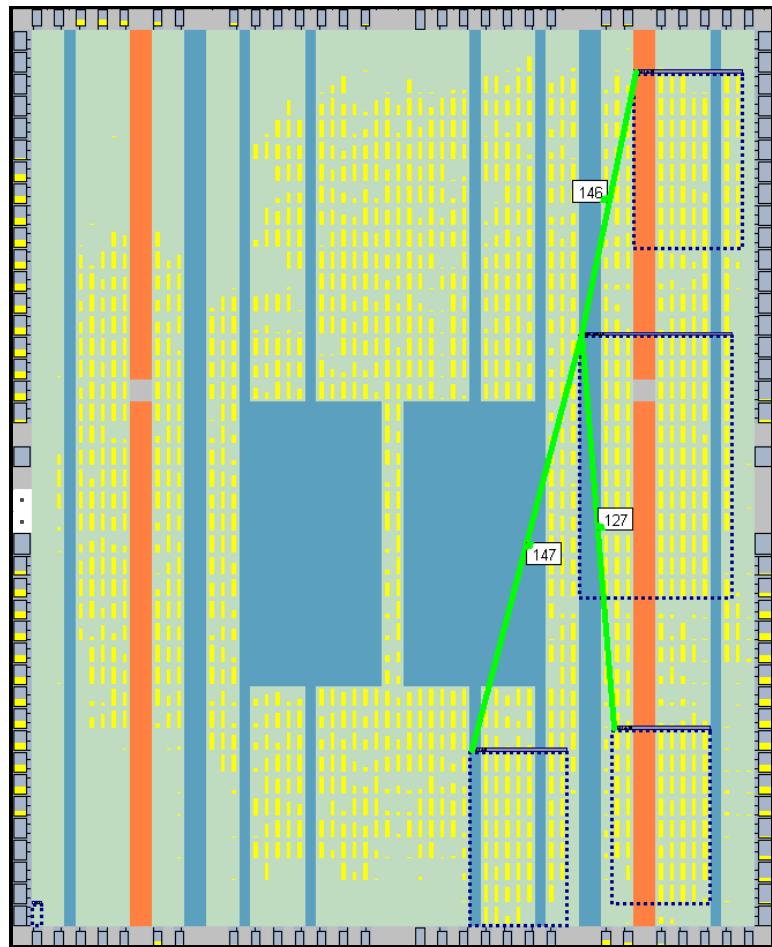
You can use the physical timing estimate information when manually placing logic in a device. This allows you to place critical nodes and modules closer together and non-critical or unrelated nodes and modules further apart. This reduces the routing congestion between critical and non-critical entities and modules allowing the Quartus II Fitter to select the timing requirements.

LogicLock Region Connectivity

You can also see how logic in LogicLock regions interface by viewing the connectivity between LogicLock regions. This capability is extremely valuable when entities are assigned to LogicLock regions. It is also possible to see the fan-in and fan-out of selected LogicLock regions.

Figure 7–11 shows standard LogicLock region connections. To view the connections in the timing closure floorplan, click the **Show LogicLock Regions Connectivity** icon in the toolbar or choose **Routing > Show LogicLock Regions Connectivity** (View menu).

Figure 7–11. LogicLock Region Connections with Connection Count

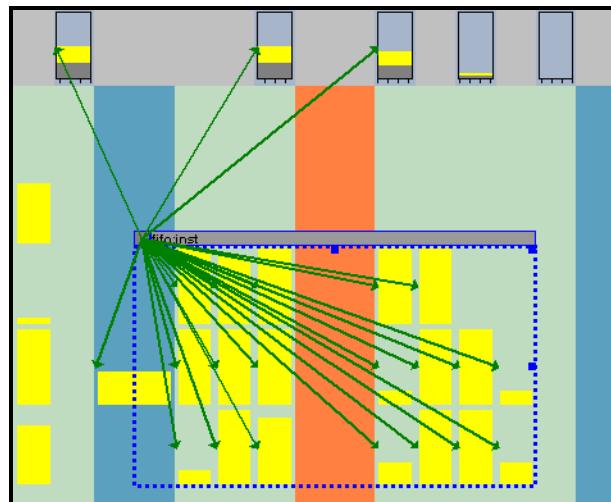


The connection line thickness indicates how many connections exist between regions. To view the number of connections between regions, click the **Show Connection Count** icon or choose **Routing > Show Connection Count** (View menu).

LogicLock region connectivity is applicable only when the user assignments are viewed in the floorplan. When you use floating LogicLock regions, the origin of the user-assigned region is not necessarily the same as the fitter-placed region. You can change the origin of your floating LogicLock regions to that of the last compilation origin in the **LogicLock Regions** window (Assignments menu), or by selecting **Back-Annotate Origin and Lock** under **Location** in the **LogicLock Regions Properties** dialog box.

To see the fan-in or fan-out of a LogicLock region, select the user-assigned LogicLock region while the fan-in or the fan-out option is turned on. To set the fan-in option, click the **Show Node Fan-In** icon or choose **Routing > Show Node Fan-In** (View menu). To set the Fan-Out option, select the **Show Node Fan-Out** icon or choose **Routing > Show Node Fan-Out** (View menu). Only the nodes that have user assignments are seen when viewing fan-in or fan-out of LogicLock regions. [Figure 7–12](#) shows the fan-out of a selected LogicLock region.

Figure 7–12. Fan-In or Fan-Out



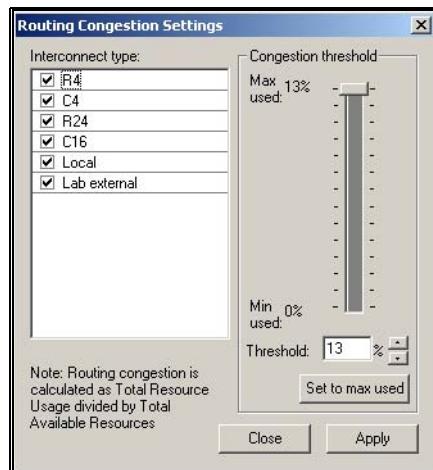
Viewing Routing Congestion

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 visually represented 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.

To view routing congestion in the floorplan, click the **Show Routing Congestion** icon, or choose **Routing > Show Routing Congestion** (View menu). To set the criteria for the critical path you wish to view, click the **View Routing Congestion Settings** icon or choose **Routing > Routing Congestion Settings** (View menu). [Figure 7-13](#) shows the **Routing Congestion Settings** dialog box.

Figure 7-13. Routing Congestion Settings Window



You can choose the routing resource 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.

If you are using the routing congestion viewer to determine where there is a lack of routing resources, examine each routing resource individually to see which ones use close to 100% of available resources.

I/O Timing Analysis Report File

Use the **Timing Analyzer** folder in the **Compilation Report**

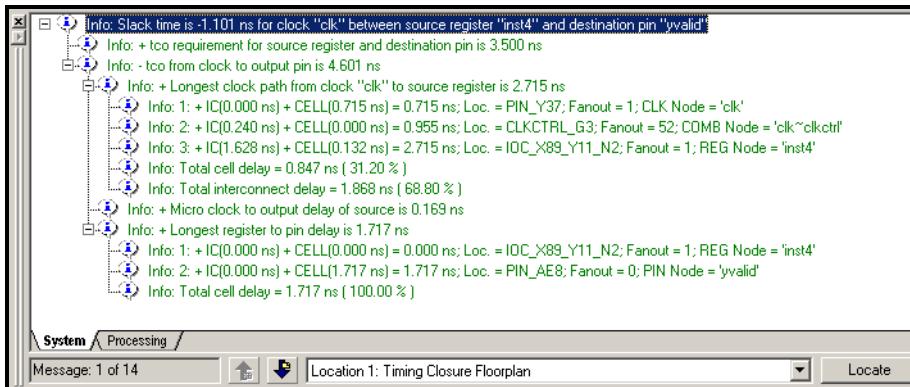
(Processing menu) to determine whether I/O timing has been met. The t_{SU} , t_H , and t_{CO} reports list the I/O paths and the slack associated with each. The I/O paths that did not meet the required timing are reported with a negative slack and are displayed in red, as shown in [Figure 7-14](#).

Figure 7-14. I/O Requirements

tco							
	Slack	Required tco	Actual tco	From	To	From Clock	
1	-1.101 ns	3.500 ns	4.601 ns	inst4	yvalid	clk	
2	0.399 ns	5.000 ns	4.601 ns	state_m:inst1 filter~:			Copy Ctrl+C
3	0.399 ns	5.000 ns	4.601 ns	inst5[7]			Select All Ctrl+A
4	0.399 ns	5.000 ns	4.601 ns	inst5[6]			
5	0.399 ns	5.000 ns	4.601 ns	inst5[5]			
6	0.399 ns	5.000 ns	4.601 ns	inst5[4]			
7	0.399 ns	5.000 ns	4.601 ns	inst5[3]			
8	0.399 ns	5.000 ns	4.601 ns	inst5[2]			
9	0.399 ns	5.000 ns	4.601 ns	inst5[1]			
10	0.399 ns	5.000 ns	4.601 ns	inst5[0]			

To determine why timing requirements are not met, right-click a particular I/O entry and choose **List Paths**. A message appears in the **System** tab of the **Message** window. You can expand a selection by clicking the “+” icon at the beginning of the line, as shown in [Figure 7-15](#). This is a good method of determining where along the path the greatest delay is located.

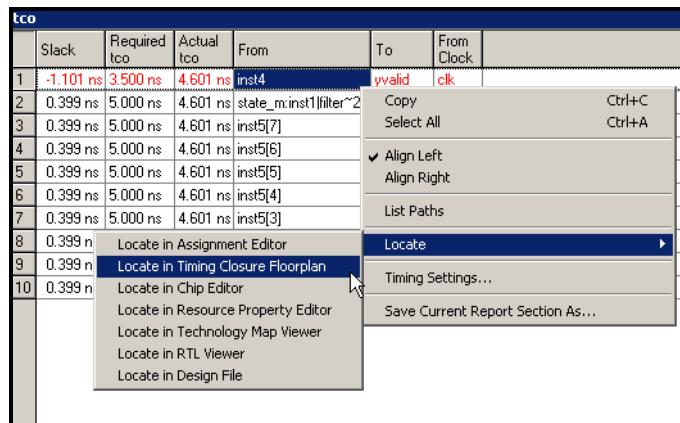
Figure 7-15. I/O Slack Report



To visually analyze I/O timing, right-click on an I/O entry in the report and select **Locate in Timing Closure Floorplan**, as shown in [Figures 7–16](#). The Timing Closure Floorplan Editor is displayed, highlighting the I/O path.

 You can set the level of detail in the floorplan in the View menu.

Figure 7–16. Locate Failing Path in Timing Closure Floorplan Editor

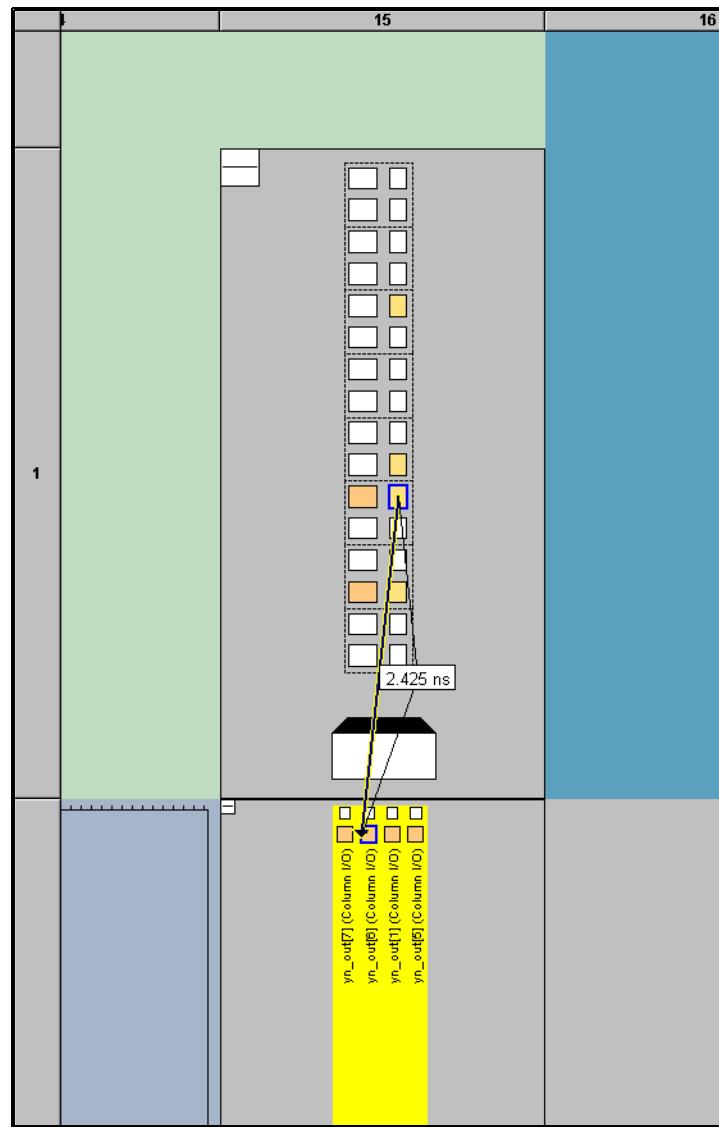


The screenshot shows a table titled "tco" with columns: Slack, Required tco, Actual tco, From, To, and From Clock. The first row has highlighted values: Slack = -1.101 ns, Required tco = 3.500 ns, Actual tco = 4.601 ns, From = inst4, To = valid, and From Clock = clk. A context menu is open over this row, listing options: Copy (Ctrl+C), Select All (Ctrl+A), Align Left, Align Right, List Paths, Locate (highlighted with a blue arrow), Timing Settings..., and Save Current Report Section As... .

	Slack	Required tco	Actual tco	From	To	From Clock
1	-1.101 ns	3.500 ns	4.601 ns	inst4	valid	clk
2	0.399 ns	5.000 ns	4.601 ns	state_m:inst1/filter^2		
3	0.399 ns	5.000 ns	4.601 ns	inst5[7]		
4	0.399 ns	5.000 ns	4.601 ns	inst5[6]		
5	0.399 ns	5.000 ns	4.601 ns	inst5[5]		
6	0.399 ns	5.000 ns	4.601 ns	inst5[4]		
7	0.399 ns	5.000 ns	4.601 ns	inst5[3]		
8	0.399 n	Locate in Assignment Editor				
9	0.399 n	Locate in Timing Closure Floorplan				
10	0.399 n	Locate in Chip Editor				
	Locate in Resource Property Editor					
	Locate in Technology Map Viewer					
	Locate in RTL Viewer					
	Locate in Design File					

In [Figure 7–17](#), the arrows indicate the critical path (i.e., a register) from the beginning point to the end point (i.e., a pin). The times shown are the slack figures for each path. Negative slack indicates paths that failed to meet their timing requirements.

Figure 7–17. Critical I/O Paths in the Timing Closure Floorplan



f_{MAX} Timing Analysis Report File

To determine whether your system performance or f_{MAX} timing requirements are met, the Quartus II software generates a timing analysis report that provides detailed timing information on every clock in your design. This report is accessed by opening the **Timing Analyzer** folder in the **Compilation Report** (Processing menu). The **Clock Setup** folder of the **Compilation Report** provides figures for slack and register-to-register f_{MAX} . The paths that are not meeting timing requirements are shown in red. See [Figure 7–18](#).

Figure 7–18. f_{MAX} Requirements

	Slack	Actual fmax (period)	From	To	From Clock	To Clock
1	2.096 ns	126.36 MHz (period = 7.914 ns)	state_minst[1]filter~28	acc:inst3[result][1]	clk	clk
2	2.115 ns	126.82 MHz (period = 7.885 ns)		Copy	Ctrl+C	sult[11]
3	2.222 ns	126.57 MHz (period = 7.778 ns)		Select All	Ctrl+A	sult[10]
4	2.222 ns	126.57 MHz (period = 7.778 ns)				sult[9]
5	2.222 ns	126.57 MHz (period = 7.778 ns)				sult[8]
6	2.222 ns	126.57 MHz (period = 7.778 ns)				sult[7]
7	2.222 ns	126.57 MHz (period = 7.778 ns)				sult[6]
8	2.318 ns	130.17 MHz (period = 7.682 ns)				Locate
9	2.343 ns	130.60 MHz (period = 7.657 ns)				sult[10]
10	2.546 ns	134.16 MHz (period = 7.454 ns)				sult[11]
11	2.557 ns	134.35 MHz (period = 7.443 ns)				Timing Settings...
12	2.578 ns	134.73 MHz (period = 7.422 ns)				Save Current Report Section As...
13	2.583 ns	134.83 MHz (period = 7.417 ns)				state_minst[1]filter~26_Duplicate_1
14	2.619 ns	135.48 MHz (period = 7.381 ns)				acc:inst3[result][9]
15	2.638 ns	135.83 MHz (period = 7.362 ns)				acc:inst3[result][11]
16	2.661 ns	136.26 MHz (period = 7.339 ns)				acc:inst3[result][10]
17	2.661 ns	136.26 MHz (period = 7.339 ns)				acc:inst3[result][9]
18	2.661 ns	136.26 MHz (period = 7.339 ns)				acc:inst3[result][8]
19	2.661 ns	136.26 MHz (period = 7.339 ns)				acc:inst3[result][7]
20	2.661 ns	136.26 MHz (period = 7.339 ns)				acc:inst3[result][6]
21	2.697 ns	136.93 MHz (period = 7.303 ns)				acc:inst3[result][10]
22	2.697 ns	136.93 MHz (period = 7.303 ns)				acc:inst3[result][9]
23	2.697 ns	136.93 MHz (period = 7.303 ns)				acc:inst3[result][8]
24	2.697 ns	136.93 MHz (period = 7.303 ns)				acc:inst3[result][7]
25	2.697 ns	136.93 MHz (period = 7.303 ns)				acc:inst3[result][6]
26	2.749 ns	137.91 MHz (period = 7.251 ns)	state_minst[1]filter~26_Duplicate_1	acc:inst3[result][7]	clk	clk
27	2.749 ns	137.91 MHz (period = 7.251 ns)	state_minst[1]filter~28	acc:inst3[result][6]	clk	clk
28	2.753 ns	137.99 MHz (period = 7.247 ns)	state_minst[1]filter~28	acc:inst3[result][4]	clk	clk
29	2.768 ns	138.27 MHz (period = 7.232 ns)	tags:instbn_1[1]*reg0	acc:inst3[result][11]	clk	clk
30	2.794 ns	138.77 MHz (period = 7.206 ns)	tags:instbn_2[0]*reg0	acc:inst3[result][11]	clk	clk

To analyze why timing was not met, right-click on a particular path reported in the **System** tab of the **Message** window (see [Figure 7–19](#)) and select **List Paths** (right button pop-up menu) to determine the location of the greatest delay along the path. You can expand a selection by clicking the “+” icon at the beginning of the line.

Figure 7–19. f_{MAX} Slack Report

	Slack	Actual fmax (period)	From	To	From Clock	To Clock	Required Setup Relationship
1	9.246 ns	173.79 MHz (period = 5.754 ns)	state_m:inst1 filter~26_Duplicate_1	acc:inst3 result[111].clk	clk	clk	15.000 ns
2	9.282 ns	174.89 MHz (period = 5.718 ns)	state_m:inst1 filter~26_Duplicate_1		Copy		Ctrl+C
3	9.318 ns	175.99 MHz (period = 5.682 ns)	state_m:inst1 filter~26_Duplicate_1		Select All		Ctrl+A
4	9.354 ns	177.12 MHz (period = 5.646 ns)	state_m:inst1 filter~26_Duplicate_1		✓ Align Left		
5	9.460 ns	180.51 MHz (period = 5.540 ns)	state_m:inst1 filter~26_Duplicate_1		Align Right		
6	9.525 ns	182.65 MHz (period = 5.475 ns)	state_m:inst1 filter~26_Duplicate_1		List Paths		
7	9.589 ns	184.81 MHz (period = 5.411 ns)	state_m:inst1 filter~26_Duplicate_1		Locate		▶
8	9.612 ns	185.60 MHz (period = 5.388 ns)		Locate in Assignment Editor			
9	9.638 ns	186.50 MHz (period = 5.362 ns)		Locate in Timing Closure Floorplan			
10	9.648 ns	186.85 MHz (period = 5.352 ns)		Locate in Chip Editor			
11	9.674 ns	187.76 MHz (period = 5.326 ns)		Locate in Resource Property Editor			
12	9.684 ns	188.11 MHz (period = 5.316 ns)		Locate in Technology Map Viewer			
13	9.710 ns	189.04 MHz (period = 5.290 ns)		Locate in RTL Viewer			
14	9.714 ns	189.18 MHz (period = 5.286 ns)		Locate in Design File			
15	9.720 ns	189.39 MHz (period = 5.280 ns)	taps:inst1 n[0]~reg0	acc:inst3 result[8].clk	clk	clk	15.000 ns
16	9.743 ns	190.22 MHz (period = 5.267 ns)	tapinst1 n[0]~reg0	acc:inst3 result[11].clk	clk	clk	15.000 ns

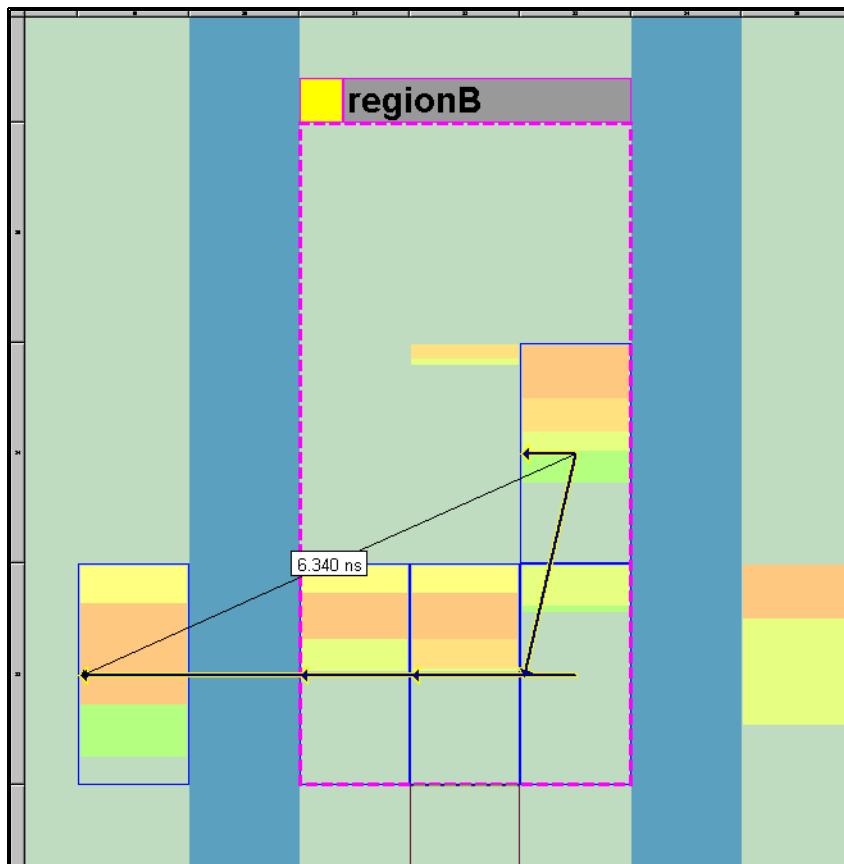
To visually analyze the f_{MAX} paths, right-click on a path in the report and select **Locate in Timing Closure Floorplan** to display the Timing Closure Floorplan Editor, which highlights the path. See Figure 7-20. Figure 7-21 shows the Timing Closure Floorplan Editor displaying a failing path.

 Double-clicking the section **Info: - Longest register to register delay is <slack value> ns** in the list path text locates the path in the Timing Closure Floorplan.

Figure 7–20. Locate Failing Path in Timing Closure Floorplan

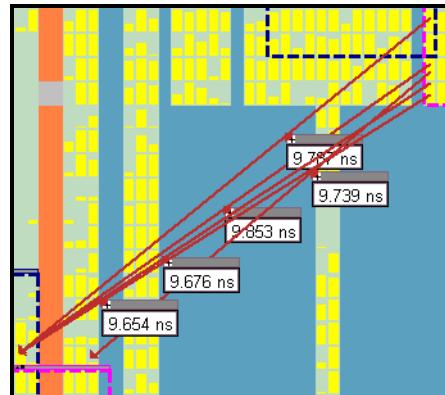
	Slack	Actual (max period)	From	To	From Clock	To Clock	Required Setup Relationship
1	9.246 ns	173.79 MHz (period = 5.754 ns)	state_m:inst1/filter~26_Duplicate_1	acc:inst3:result[111].clk	clk	15.000 ns	
2	9.282 ns	174.89 MHz (period = 5.718 ns)	state_m:inst1/filter~26_Duplicate_1	Copy			Ctrl+C
3	9.318 ns	175.99 MHz (period = 5.682 ns)	state_m:inst1/filter~26_Duplicate_1	Select All			Ctrl+A
4	9.354 ns	177.12 MHz (period = 5.646 ns)	state_m:inst1/filter~26_Duplicate_1	✓ Align Left			
5	9.460 ns	180.51 MHz (period = 5.540 ns)	state_m:inst1/filter~26_Duplicate_1	Align Right			
6	9.525 ns	182.65 MHz (period = 5.475 ns)	state_m:inst1/filter~26_Duplicate_1	List Paths			
7	9.589 ns	184.81 MHz (period = 5.411 ns)	state_m:inst1/filter~26_Duplicate_1	Locate in Assignment Editor	Locate		
8	9.612 ns	185.60 MHz (period = 5.388 ns)	Locate in Timing Closure Floorplan	Timing Settings...			
9	9.638 ns	186.50 MHz (period = 5.362 ns)	Locate in Chip Editor	Save Current Report Section As...			
10	9.648 ns	186.85 MHz (period = 5.352 ns)	Locate in Resource Property Editor				
11	9.674 ns	187.76 MHz (period = 5.326 ns)	Locate in Technology Map Viewer	acc:inst3:result[9].clk	clk	clk	15.000 ns
12	9.684 ns	188.11 MHz (period = 5.316 ns)	Locate in RTL Viewer	acc:inst3:result[9].clk	clk	clk	15.000 ns
13	9.710 ns	189.04 MHz (period = 5.290 ns)	Locate in Design File	acc:inst3:result[11].clk	clk	clk	15.000 ns
14	9.714 ns	189.18 MHz (period = 5.286 ns)					
15	9.720 ns	189.39 MHz (period = 5.280 ns)	taps:inst1[xn[0]]~reg0	acc:inst3:result[8].clk	clk	clk	15.000 ns
16	9.743 ns	190.22 MHz (period = 5.267 ns)	locinst1[inst1]	acc:inst3:result[11].clk	clk	clk	15.000 ns

Figure 7–21. Failing Path in Timing Closure Floorplan



You can view all failing paths in the Timing Closure Floorplan Editor using the **Show Critical Paths** feature. [Figure 7–22](#) shows critical f_{MAX} paths in the Timing Closure Floorplan Editor.

Figure 7–22. Critical Paths in the Timing Closure Floorplan Editor



The *Design Optimization for Altera Devices* chapter in Volume 2 of the *Quartus II Handbook* shows you how to optimize your design in the Quartus II software. With the options and tools available in the Timing Closure Floorplan and the techniques described in that chapter, the Quartus II software can assist you in achieving timing closure in a more time efficient manner.

Conclusion

Design analysis for timing closure is a fundamental requirement for optimal performance in highly complex designs. The Quartus II Timing Closure Floorplan Editor assists in closing 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.

qii52007-2.1

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 vary; 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 (.edf) or a Verilog Quartus Mapping (.vqm) file 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 following sections “[Synthesis Netlist Optimizations](#)” on page 8–2 and “[Physical Synthesis Optimizations](#)” on page 8–9 explain how the available optimizations work. This chapter also provides information on 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 on using these attributes during synthesis in the Quartus II software, see the *Quartus II Integrated Synthesis* chapter in Volume 1 of the *Quartus II Handbook*.

Synthesis Netlist Optimizations

You can view and modify the synthesis netlist optimization options in the **Synthesis Netlist Optimizations** page under **Analysis & Synthesis Settings** in the **Settings** dialog box (Assignments menu).

The sections “[WYSIWYG Primitive Resynthesis](#)” and “[Gate-Level Register Retiming](#)” on page 8–4 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 EDIF or VQM file generated by a third-party synthesis tool. This option can be found on the **Synthesis Netlist Optimizations** page under **Analysis & Synthesis Settings** in the **Settings** dialog box (Assignments menu). 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 the Stratix® II, Stratix, Stratix GX, Cyclone™ II, Cyclone, MAX® II, or APEX™ device families.

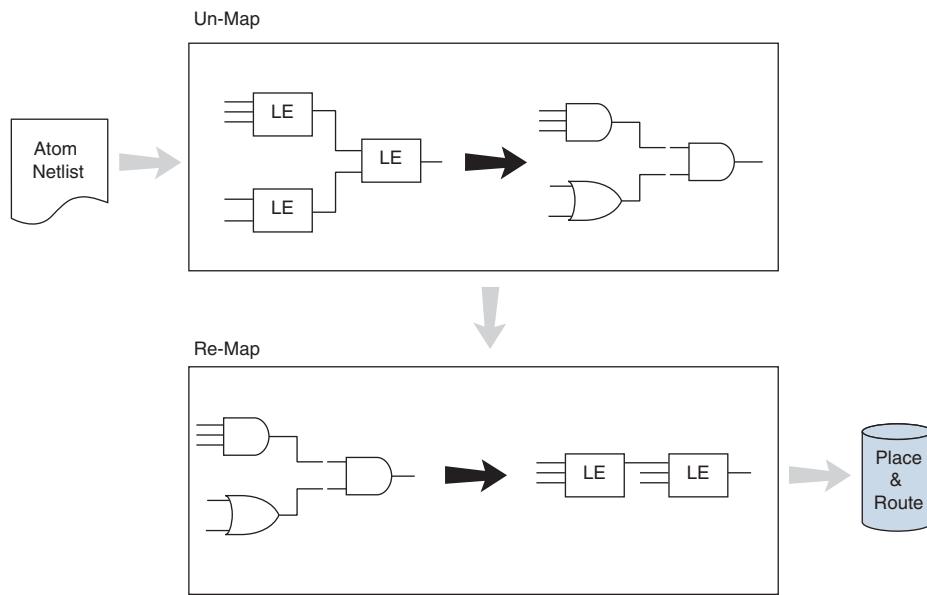
The **Perform WYSIWYG primitive resynthesis** option directs the Quartus II software to un-map the logic elements (LEs) in an atom netlist to logic gates, and then re-map the gates back to Altera-specific primitives. This feature allows the Quartus II software to use different techniques specific to the device architecture during the re-mapping process. The Quartus II technology mapper optimizes the design for **Speed**, **Area**, or **Balanced**, according to the setting of the **Optimization Technique** option on the **Analysis & Synthesis Settings** page in the **Settings** dialog box (Assignments menu). 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.



See the *Quartus II Integrated Synthesis* chapter in Volume 1 of the *Quartus II Handbook* for details on the Optimization Technique option.

Figure 8–1 shows the Quartus II software flow for this feature.

Figure 8–1. WYSIWYG Primitive Resynthesis



The **Perform WYSIWYG primitive resynthesis** option is not applicable if you are using Quartus II integrated synthesis. With 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 VQM or EDIF 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 VQM or EDIF atom netlist from your third-party synthesis tool, because the primitives in the 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.

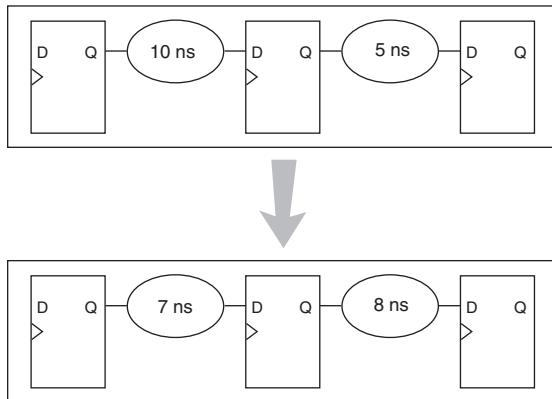
Any nodes or entities that have the **Netlist Optimizations** logic option set to **Never Allow** are not affected during WYSIWYG primitive resynthesis. This logic option can be applied with the **Assignment Editor** (Assignments menu) if you want to disable WYSIWYG resynthesis for parts of your design.

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 8–2](#) for an example. This option can be used with the Stratix II, Stratix, Stratix GX, Cyclone II, Cyclone, MAX II, APEX II, APEX 20K, or APEX 20KE device families. The option is found on the **Synthesis Netlist Optimizations** page under **Analysis & Synthesis Settings** in the **Settings** dialog box (Assignments menu).

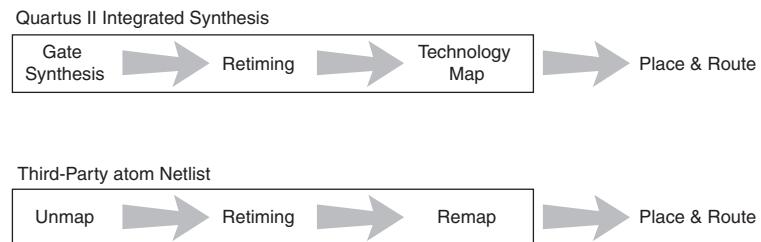
The functionality of your design is not changed when the **Perform gate-level register retiming** option is turned on. However, if any registers in your design have the **Power-Up Don't Care** logic option assigned, the values of registers during power-up may change due to this register and logic movement. The **Power-Up Don't Care** logic option is turned on globally by default. You can change the default setting for the option on the **Analysis & Synthesis Settings** page in the **Settings** dialog box (Assignments menu) by clicking **More Settings**. You can also set the logic option for individual registers or entities using the Assignment Editor. Registers that are explicitly assigned power-up values are not combined with registers that have been explicitly assigned other values.

[Figure 8–2](#) 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 8-2. 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.

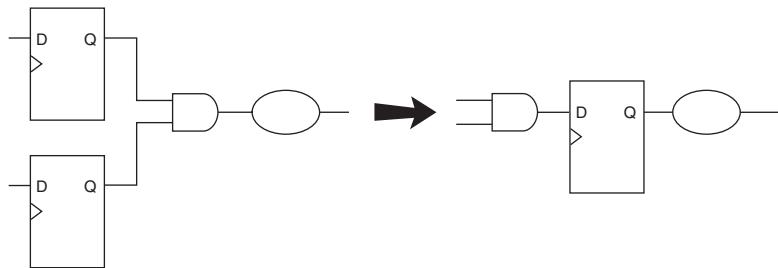
The design flows for the case of integrated Quartus II synthesis and a third-party atom netlist are shown in [Figure 8-3](#).

Figure 8-3. Gate-Level Synthesis

The gate-level register retiming options 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 8–4](#).

Figure 8–4. 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:

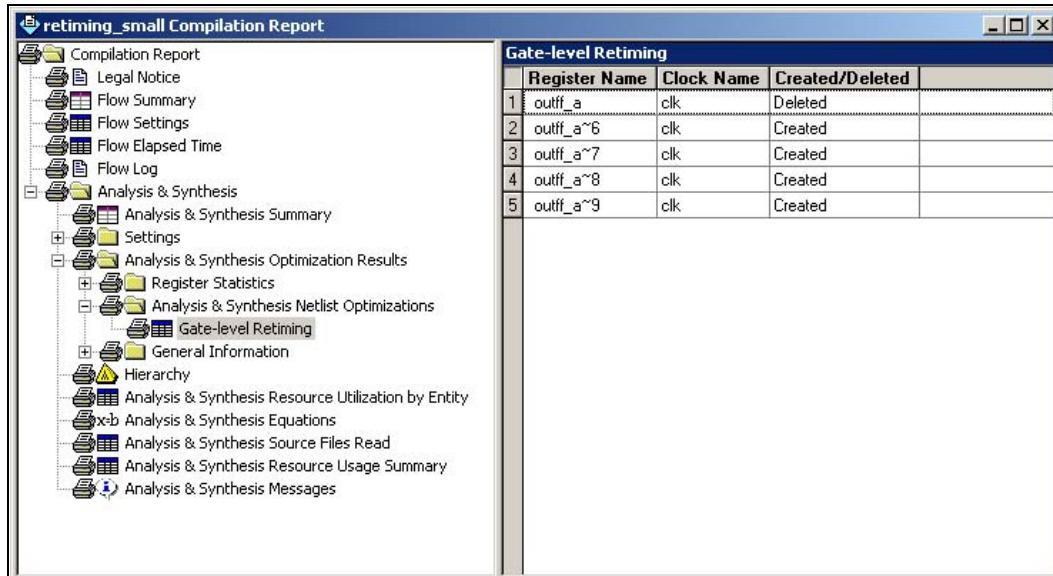
- All registers have the same clock signal
- All registers have the same clock enable signal
- All registers have asynchronous control signals that are active under the same conditions
- Only one register has an asynchronous load other than V_{CC} or GND

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. This report can be found in the **Analysis & Synthesis Netlist Optimizations** section of the **Analysis & Synthesis Optimization Results** folder under **Analysis & Synthesis** in the **Compilation Report** (Processing menu). See [Figure 8–5](#).



The node names for these registers change during the retiming process.

Figure 8–5. 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 and is applied through the Assignment Editor (Assignments menu).

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 connection from input pin-to-register-to-register connection if both registers have the same clock and the first register does not fan out to anywhere else (since these are considered synchronization registers).

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}

The **Allow register retiming to trade off t_{SU}/t_{CO} with f_{MAX}** option on the **Synthesis Netlist Optimizations** page under **Analysis & Synthesis Settings** in the **Settings** dialog box (Assignments menu) determines whether the Quartus II compiler should attempt to increase f_{MAX} at the expense of t_{SU} or t_{CO} times. 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 t_{SU}/t_{CO} with f_{MAX}** 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, you may skip the synthesis stage of the compile by avoiding running **Analysis & Synthesis**, or **quartus_map**, and instead just running the Fitter or another desired Quartus II executable.

However, if you wish, you may preserve the netlist resulting from netlist optimizations. Preserving the netlist may be required if you use the LogicLock™ flow to preserve placement and/or import one design into another. If you are using any Quartus II synthesis netlist optimization options, you can save your optimized results by turning on 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 (Assignments menu). This option saves your final results as an atom-based netlist in VQM format. By default, the Quartus II

software places the VQM file in the **atom_netlists** directory under the current project directory. If you'd like to create a different VQM file using different Quartus II settings, you may do so by changing the **File name** setting on the **Compilation Settings Process** page in the **Settings** dialog box (Assignments menu).

If you are using the synthesis netlist optimizations (and not any physical synthesis optimizations), generating a VQM file is optional. You can lock down the location of all logic and device resources in the design using the **Back-Annotate Assignments** command (Assignments menu) with or without a Quartus II-generated VQM file. Altera recommends against using 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 VQM file and wish to recompile the design, use the new VQM 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 the Stratix II, Stratix, Stratix GX, Cyclone II, and Cyclone device families.

You can view and modify the physical synthesis optimization options on the **Physical Synthesis Optimizations** page in the **Fitter Settings** section of the **Settings** dialog box (Assignments menu).

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
- 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.

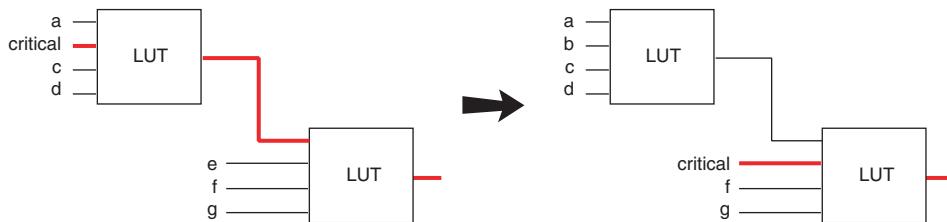
All the Physical Synthesis optimizations write results to the **Netlist Optimizations** report under **Fitter** in the **Compilation Report** (Processing menu). This report provides a list of atom netlist files that were modified, created, and deleted during physical synthesis.

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. This logic option can be applied with the Assignment Editor (Assignments menu) if you want to disable physical synthesis optimizations for parts of your design.

Physical Synthesis for Combinational Logic

The **Perform physical synthesis for combinational logic** option on the **Physical Synthesis Optimizations** page in the **Fitter** section of the **Settings** dialog box (Assignments menu) allows the Quartus II Fitter to resynthesize the design to reduce delay along the critical path. Physical Synthesis can achieve 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 8–6](#) for an example. This option also allows the duplication of LUTs to enable further optimizations on the critical path.

Figure 8–6. 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 only affects 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:

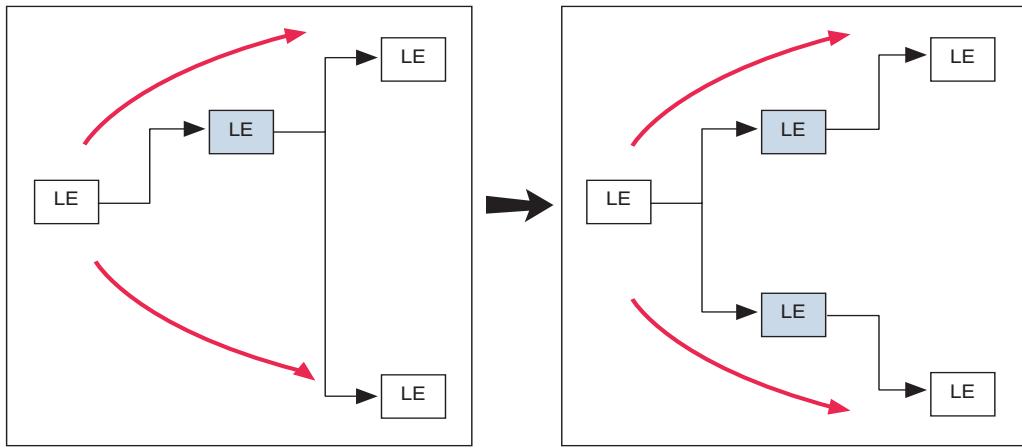
- Are part of a chain
- Drive global signals
- Are constrained to a single logic array block (LAB) location
- Have the **Netlist Optimizations** option set to **Never Allow**

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** filter 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 8–7 shows an example of register duplication.

Figure 8-7. 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 on 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 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 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 [8–4](#)). This option applies to the atom level (registers and combinational logic have already been placed into logic cells), and it complements 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 on 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 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 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.

If you are using any Quartus II physical synthesis optimization options, you can save your optimized results 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 (Assignments menu). This option saves your final results as an atom-based netlist in VQM file format. By default, the Quartus II software places the VQM file in the **atom_netlists** directory under the current project directory. If you want to create a different VQM 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 (Assignments menu).

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 (Assignments menu), a VQM 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.

Altera recommends against using a Quartus II-generated VQM 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 VQM.

To back-annotate logic locations for a design that was compiled with physical synthesis optimizations, first create a VQM. When recompiling the design with the hard logic location assignments, use the new VQM file as the input source file and turn off the physical synthesis optimizations for the new compilation.

If you are importing a VQM 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.

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 (VQM or EDIF), the **WYSIWYG Primitive Resynthesis** option must be turned on in order to use the **Gate-level Register Retiming** option.

A 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 on using the DSE script to run multiple compilations, see the *Design Space Explorer* chapter in Volume 2 of the *Quartus II Handbook*. For information on typical performance results using combinations of netlist optimization options and other optimization techniques, see the *Design Optimization for Altera Devices* 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.

For detailed information about specific scripting command options and Tcl API packages, type `quartus_sh --qhelp` at a system command prompt to run the Quartus II Command-Line and Tcl API Help utility.



For more information on Quartus II scripting support, including examples, see the *Tcl Scripting* and *Command-Line Scripting* chapters 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 8-1 lists the Quartus II Settings File (QSF) variable name and applicable values for the settings discussed in “[Synthesis Netlist Optimizations](#)” on page 8-2. 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 8-1. Synthesis Netlist Optimizations & Associated Settings

Setting Name	QSF 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 f_{MAX}	ADV_NETLIST_OPT_RETIRE_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>	<filename>	
Allow Netlist Optimizations	ADV_NETLIST_OPT_ALLOWED	"ALWAYS ALLOW", "DEFAULT", "NEVER ALLOW"	Instance

Physical Synthesis Optimizations

Table 8–2 lists the QSF variable name and applicable values for the settings discussed in “Physical Synthesis Optimizations” on page 8–9. 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 8–2. Physical Synthesis Optimizations and Associated Settings			
Setting Name	QSF Variable Name	Values	Type
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
Power-Up Don't Care	ALLOW_POWER_UP_DONT_CARE	ON, OFF	Global
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>	

Back-Annotating Assignments

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 on back-annotating assignments, see “Preserving Synthesis Netlist Optimization Results” on page 8–8 or “Preserving Your Physical Synthesis Results” on page 8–13.

The following Tcl command back-annotes 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.

qii52008-2.1

Introduction

The Quartus® II software includes many advanced optimization algorithms to help you achieve timing closure. The various settings and parameters control the behavior of the algorithms. These options provide complete control over the Quartus II software optimization techniques.

Because each FPGA design is unique, there is no standard set of options that always results in the best performance. 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 the process of 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 & Exploration Point

Before a design is explored by DSE, an exploration space is created. An exploration space is comprised of various synthesis and Fitter settings that are available in the Quartus II software. A single group of settings in the exploration space is referred to as a *point*. DSE traverses the points in an exploration space to determine the optimal settings for your design.

Seed & Seed Sweeping

The Quartus II Fitter utilizes a seed that specifies the starting value that randomly determines the initial placement for the current design. The value of the seed can be any non-negative integer value. Changing the starting value may or may not produce better fitting. By varying the value of the seed or seed sweeping, an optimal value can be determined for the current design.

DSE extends the concept of Fitter seed sweeping with exploration spaces, providing a method for sweeping through general compilation and Fitter parameters to find the best options for your design. You can run DSE in a variety of exploration modes, ranging from an exhaustive try-all-options-and-values mode to one that focuses on one parameter.

DSE Exploration

DSE compares all exploration space point results with the results of a base compilation. This base compilation result is generated from the initial settings that were specified in the original Quartus II project files. As DSE traverses all points in the exploration space, all settings that are not explicitly modified by DSE default to the base compilation setting. For example, if an exploration space point turns on register retiming and does not modify the register packing setting, the register packing setting defaults to the value specified in the base compilation.



The base compilation is performed with the settings of 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 graphical user interface (GUI) or command-line mode. To run DSE with the GUI, either type `quartus_sh --dse ↵` at a command prompt or select **Design Space Explorer** (Tools menu).

To run DSE in command-line mode, type the following at the command prompt:

```
quartus_sh --dse -nogui [<options>] ↵
```

DSE can be run with the following options:

```
-project <project name>
-revision <revision name>
-seeds <seed list>
-llr-restructuring
-exploration-space <space>
-optimization-goal <goal>
-search-method <method>
-custom-file <filename>
-stop-after-time <stop-after-time value>
-ignore-failed-base
-archive
-run-assembler
-slaves <slave list>
-use-lsf
-slack-column <column name>
-nogui
-ignore-signalprobe
-ignore-signaltap
-help
```

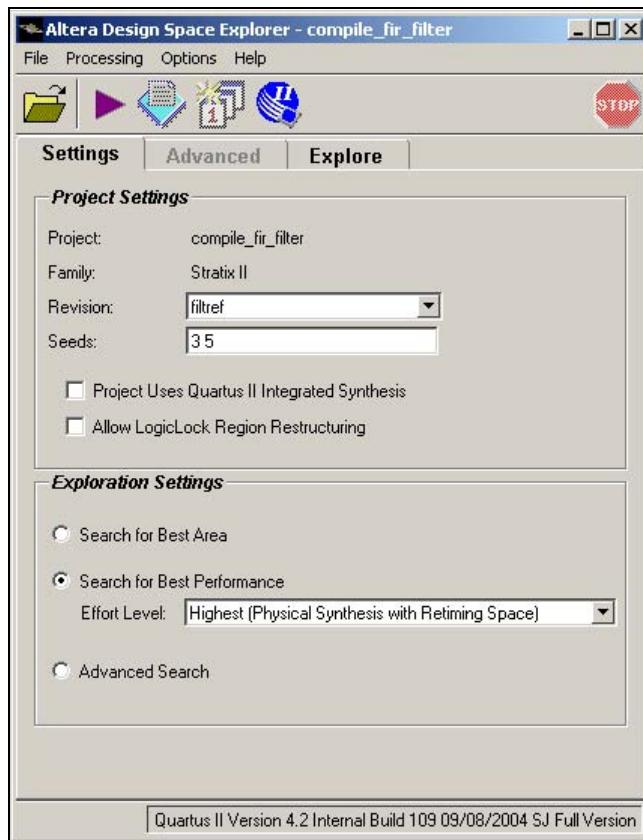
The DSE script is in the default Quartus II software installation at <Quartus II install directory>\bin\tcl_scripts\dse\dse.tcl on the PC platform and <Quartus II install directory>/<platform>/tcl_scripts/dse/dse.tcl on the Solaris, HP-UX, and Linux platforms.



For more information, type quartus_sh --help=dse ↵ at the command prompt.

Figure 9–1 shows the DSE user interface. The Settings tab is divided into two sections: **Project Settings** and **Exploration Settings**.

Figure 9–1. DSE User Interface



DSE Flow

You can run DSE at any point in the design process. However, Altera recommends that you run DSE very late in your design cycle when you are focussed on increasing the performance of the design. The results gained from different combinations of optimization options may not persist over large changes in a design. You can run DSE in signature mode at the midpoint in your design cycle to see the effect of various parameters such as the register packing logic option.

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 upon the **Optimization Goal** selected for the exploration. For example, if the **Optimization Goal** is set to **Optimize for Speed**, the Quartus II software tries to achieve all your timing requirements and DSE reports the compilation with the smallest slack. Therefore, it is important that you correctly specify all timing requirements in your Quartus II project before performing a design exploration with DSE.

You can change the initial placement configuration used by the Quartus II Fitter by varying the **Fitter Seed** value. You can enter seeds in the **Seeds** field of the DSE user interface.



When using the Quartus II software, the seed value is set in the **Fitter Settings** page of the **Settings** dialog box (Assignments menu).

Compilation time increases as DSE exploration spaces become more comprehensive. This increase in compilation time comes as a result of running several compilations and comparing the reported slack with the original 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

The following device families support all **Advanced Exploration Space** types:

- Stratix® II
- Stratix
- Stratix GX
- Cyclone™ II
- Cyclone
- MAX® II

The following device families support only the **Advanced Exploration Support** settings shown in [Table 9-1](#):

- APEX™ 20K
- APEX 20KC
- APEX 20KE
- APEX II
- FLEX® 10K
- FLEX 10KA
- FLEX 10KE

Table 9–1. Advanced Exploration Space Support for APEX 20K, APEX II, & FLEX 10K Devices

Seed sweep	Area optimization space
Signature fitting effort level	Extra effort space
Extra effort for Quartus Integrated Synthesis Projects	Custom space

The following device families support the **Synthesis Space** type:

- MAX 3000A
- MAX 7000AE
- MAX 7000B
- MAX 7000S



The **Synthesis Space** type support is available only at the command line.

DSE Project Settings

This section provides the following DSE project setting information:

- Setting up the DSE working environment
- Specifying the revision
- Setting the initial seed
- Restructuring LogicLock regions
- Effort Search for Quartus Integrated Synthesis Projects

Setting Up the DSE Working Environment

The DSE user interface provides two methods to open a Quartus II project for a design exploration. By selecting **Open Project** (File menu) you can browse to your project. You can also use the **Open** icon to open a project for a design exploration.

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 once the Quartus II project has been opened.



If no revisions are 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

You specify the seed that DSE uses for an exploration 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. DSE uses every seed in the range.

Restructuring LogicLock Regions

If your design is written in VHDL or Verilog HDL, turn on the **Project Uses Quartus II Integrated Synthesis** option to allow DSE to explore synthesis options.

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 may remove LogicLock regions that negatively affect the performance of the design.

Use the **Exploration Settings** list to select the type of exploration to perform: **Search for Best Area**, **Search for Best Performance**, or **Advanced Search**.



The “**Exploration Space**” section on page 9–8 describes the type of explorations you can perform.

Search for Best Performance and Search for Best Area Options

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 Best Area** option uses a predefined exploration space that targets device utilization improvements for your design.

Advanced Search Options

Advanced Search options provide full control over the exploration space, the optimization goal for your design, and the search method used in a design exploration. The section titled “[Performing an Advanced Search in Design Space Explorer](#)” on page 9–8 provides detailed information on how to set up and perform an advanced search in DSE.



The advanced search can be used to define equivalent exploration spaces to those found in the **Search for Best Area** and **Search for Best Performance** options.

Effort Search for Quartus Integrated Synthesis Projects

The **Effort Search for Quartus Integrated Synthesis Projects** exploration space works only for designs that have been synthesized using the Quartus II integrated synthesis. With this option enabled, DSE explores options that affect the synthesis stage of compilation.

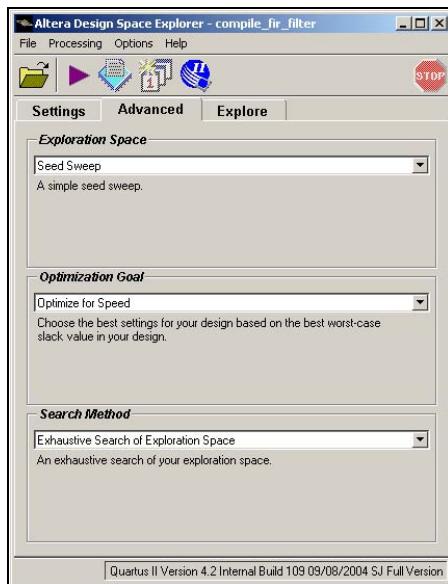


For more information on integrated synthesis options, see the *Quartus II Integrated Synthesis* chapter in Volume 1 of the *Quartus II Handbook*.

Performing an Advanced Search in Design Space Explorer

You must make three exploration settings in the **Advanced Search** dialog box before exploring a design space. These three settings, **Exploration Space**, **Optimization Goal**, and **Search Method**, are described in the following sections. [Figure 9-2](#) shows the **Advanced Search** dialog box.

Figure 9-2. DSE Advanced Search Dialog Box



Exploration Space

The exploration space list controls the type of exploration that DSE performs on your design. DSE traverses the points in an exploration space, applying the settings to the design and comparing the compilation results to determine the best settings for your design. DSE offers the following types of exploration spaces:

- Seed sweep
- Extra effort search
- Physical synthesis search
- Retiming space
- Area optimization search
- Custom space
- Signature mode



Not all advanced exploration spaces are available for every device family. See “[DSE Support for Altera Device Families](#)” on page 9–5 for advanced exploration space support for various device families.

Compilation time increases proportionally to the breadth of the exploration; the design space increases as more optimization options and parameters are explored.



The **Exploration Space** field is enabled after a project has been opened in DSE.

Turn on **Save exploration space to file** (Option menu) to save an XML file representing the exploration space. The exploration space is written 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, see “[Creating Custom Spaces for DSE](#)” on page 9–17.

Seed Sweep

The Seed Sweep exploration space leverages the seed sweeping concept and automates the process. Enter the seed values in the **Seeds** field in the DSE user interface. There are no “magic” seeds. Because the variation between seeds is truly random, any 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 make changes to your netlist.



The seed field accepts individual seed values, for example, 2, 3, 4, and 5, or seed ranges, for example, 2-5.

There is a 1× increase in compilation time for every seed value specified. For example, if you enter five seeds, the compilation time is 5× the initial compilation time.

Extra Effort Space

The **Extra Effort Search** exploration space adds the **Register Packing** option to the exploration space performed by the **Seed Sweep**. This exploration type also increases the Quartus II Fitter effort during the place-and-route stage. This type of exploration makes no changes to your netlist.

Physical Synthesis Search

The **Physical Synthesis Search** exploration space adds physical synthesis options such as register retiming and physical synthesis for combinational logic to the options included in the **Extra Effort Search** exploration space. These netlist optimizations move registers in your design. Look-up tables (LUTs) may be modified. The design behavior is not affected by these options.



For more information about physical synthesis, see the *Netlist Optimization & Physical Synthesis* chapter in Volume 2 of the *Quartus II Handbook*.

The **Physical Synthesis for Quartus 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 Integrated Synthesis Projects** exploration space works only for designs that have been synthesized using Quartus II integrated synthesis.

Retiming Search

The **Physical Synthesis with Retiming Search** exploration space includes all the options in the **Physical Synthesis Search** exploration space and explores register retiming. The register retiming may move registers in your design.

The **Retiming Search for Quartus Integrated Synthesis Projects** exploration space includes all the options in **Retiming Search** exploration space, and explores various Quartus II integrated synthesis optimization options. The **Retiming Search for Quartus integrated synthesis Projects** exploration space works only for designs that have been synthesized using the Quartus II integrated synthesis.

Area Optimization Search

The **Area Optimization Search** exploration space explores options that affect logic cell utilization for your design. These options include register packing and **Optimization Technique** set to **Area**.

Custom Space

Use the **Custom Space** exploration space to selectively explore the effects of various optimization options on your design. This exploration type 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.

For a summary of the settings adjusted by each exploration space, refer to [Table 9–2](#).

Table 9–2. Summaries of Exploration Spaces <i>Note (1)</i>						
Search Type	Exploration Spaces					
	Seed Sweep	Extra Effort	Physical Synthesis	Retiming	Area Optimization	Custom
Analysis & 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 9–2:

- (1) For exploration spaces that include Quartus Integrated Synthesis, DSE increases the synthesis effort.



For more information about using custom exploration spaces with DSE, see “[Creating Custom Spaces for DSE](#)” on page 9–17.

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 when exploring the design space of your design whether it be performance, logic utilization, or a combination of both. These metrics allow you to selectively determine which compilation is best, based on the requirements of the design. DSE uses the **Optimization Goal** setting to determine the best compilation results. Here you can specify to DSE which optimization goal you are trying to achieve. [Table 9–3](#) summarizes the four available optimization settings.

Table 9–3. Optimization Goal Settings

Setting	Description
Optimize for speed	The exploration space point that contains the best worst-case slack value is selected by DSE as the best run
Optimize for area	The exploration space point that contains the lowest logic cell count is selected by DSE as the best run
Optimize for failing paths	The exploration space point that contains the least amount of failing paths is selected by DSE as the best run
Optimize for negative slack and failing path	The exploration space point that contains the best average negative worst-case slack and lowest number of failing paths is selected by DSE as the best run

Search Method

The **Search Method** setting allows you to control the breadth of the search performed by DSE. DSE provides three search methods: **exhaustive search of exploration space**, **accelerated search of exploration space**, and **distributed search of exploration space**. These three search methods are described in [Table 9–4](#).

Table 9–4. 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 the optimal settings for your design, but 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.
Distributed search of exploration space	Equivalent to the exhaustive search of exploration space except that this search method uses cluster computing technology to decrease DSE run time.

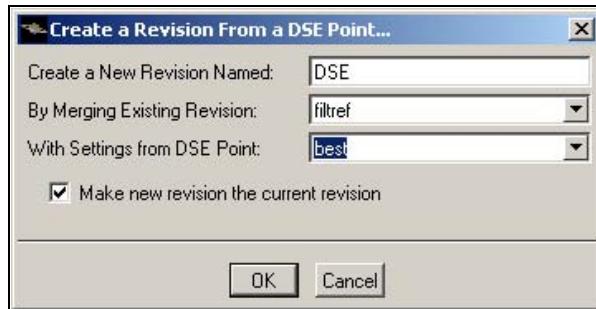
DSE Flow Options

You can control the design exploration run time with the following options:

- Create a Revision from a DSE Point
- Continue Exploration Even if Base Compilation Fails
- Run Quartus Assembler During Exploration
- Archive All Compilations
- Save Exploration Space to File
- Stop Flow After Time

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 space point. This option facilitates the creation of multiple revisions based on the same space point for further optimization within the Quartus II software. [Figure 9–3](#) shows the **Create a Revision From a DSE Point** dialog box.

Figure 9–3. Create a Revision from a DSE Point

The criterion 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 space point will change according to the inequality:

$$<\text{Current Decision Column Value}> > <\text{Previous Decision Column Value}>$$

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 by choosing **Advanced > Change Decision Column** (Options menu). Table 9–5 lists the available decision columns. The decision column can be any column within the Quartus II Timing Analyzer Report.

Table 9–5. DSE Change Decision Columns

Decision Column Name	Description
Worst-case slack (default)	Determines best space point on worst-case slack in the exploration space
Clock slack: <i><clock name></i>	Determines best space point on the <i><clock name></i> specified
Clock slack: *	Determines best space point on all clocks
Worst-case minimum t_{CO} (slack)	Determines best space point on worst case-case minimum t_{CO} slack
Worst-case t_H (slack)	Determines best space point on worst-case t_H slack
Worst-case t_{SU} (slack)	Determines best space point on worst case-case t_{SU} slack
<i><any column name></i>	Determines best space point on any column available in the Quartus II timing analysis report file

Continue Exploration Even if Base Compilation Fails

With this option enabled, DSE will continue 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 this option on.

Run Quartus Assembler During Exploration

By default, DSE does not generate programming files for each compilation during exploration. Turn on **Run Quartus Assembler During Exploration** to generate programming files for each point DSE determines as the best exploration point. This will result in DSE generating programming files only for an exploration point that is considered the best as compared to any previous exploration point in the current exploration.

Archive All Compilations

Turn on **Archive All Compilations** to create a Quartus Archive File (**.qar**) for each compilation. These archive files are saved to the **dse** directory in the design's working directory.

Save Exploration Space to File

Turn on **Save Exploration Space to File** to write out a *<project name>.dse* file that contains all options explored by DSE. You can use or modify this file to perform a custom exploration.

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.

DSE Advanced Information

This section covers advanced features that are available in DSE. These features are made available to increase the processing efficiency of design space exploration as well as to provide further customization of the design space.

Computer Load Sharing in DSE Using Distributed Exploration Searches

When the **Search Method** is set to **Distributed Search of Exploration Space**, DSE uses cluster computing technology to decrease exploration search time. DSE uses multiple client computers to compile points in the specified exploration space. Two modes of operation are available when using the **Distributed DSE** option. The first mode uses the Platform LSF grid computing technology to distribute exploration space points to a computing network. In the second mode, DSE acts as a master and distributes exploration space points to client computers. Both modes use an **Exhaustive Search of Exploration Space** search method.

Distributed DSE Using LSF

The easiest way to use distributed DSE technology is to submit the compilations to a pre-configured LSF cluster at your local site. For more information on LSF software, see www.platform.com, or contact your system administrator. Turn on **Use LSF resources** to enable this feature.

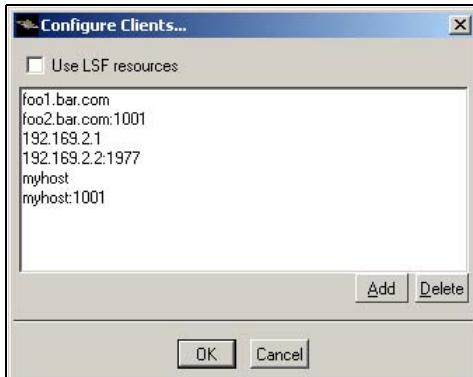
Distributed DSE Using a Quartus II Master Process

Before DSE can use machines in the local area network to compile points in the exploration space, you need to create Quartus II software slave instances on the machines. In most cases, creating a slave instance on a machine is simple. Type the following command at a command prompt on a client machine:

```
quartus_sh --qslave ↵
```

Repeating this on several machines creates a cluster of Quartus II software slaves for DSE to use. Once you have created a set of Quartus II software slaves on the network, add the names of each slave machine in the **Enter Clients** dialog box. This dialog box appears after selecting **Exhaustive Search of Exploration Space**. Figure 9–4 shows an example of client entries for a distributed search.

Figure 9–4. 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 on running and configuring Quartus slaves, type `quartus_sh --help=qslave` ↵ at the command prompt.

The version of the Quartus II software that you use for the Quartus II software slaves must be the same as the version of the Quartus II software you use to run DSE. To see the version of the Quartus II software you are using to run DSE, choose **About DSE** (Help menu). Unexpected results can occur if you mix Quartus II software versions when using the Distributed DSE search feature.

Creating Custom Spaces for DSE

You can use custom spaces to explore combinations of options that are outside the predefined exploration spaces in the Exploration Space list. An exploration space is defined in an XML file. The following is a description of the tags used to create a custom space for DSE to process.

A custom space is defined by three pairs of major tags, which are:

- <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 is </DESIGNSPACE>. This tag defines the end of the exploration space. These are both required tags 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 space point of a custom space. The end tag is </POINT>. This tag defines the end of the exploration space point. The POINT also allows you to specify “stage” 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**—indicating an Analysis & Synthesis setting change for that particular point
- **fit**—indicating a Fitter setting change for that particular point
- **seed**—indicating a Fitter seed change
- **llr**—indicating a LogicLock property change

The <value> value in the POINT tag can either be “1,” indicating that the exploration space point is active, or “0” for an inactive point. An example of a POINT tag is as follows:

```
<POINT space="map" enabled="1">
...
</POINT>
```

The preceding point indicates a point that has Analysis and Synthesis setting changes and is active.

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 that particular exploration space point. The end tag is `</PARAM>`. This tag defines the end of the parameter. The Analysis & Synthesis settings and the "*<parameter>*" values are shown in Table 9–6. Table 9–7 shows the Fitter settings. An example of a PARAM tag is shown below:

```
<PARAM name="ADV_NETLIST_OPT_SYNTH_GATE_RETIME"> ON
</PARAM>
```

The point in the example above indicates that the Analysis and Synthesis setting gate-level retiming is turned on for the exploration space point.

Table 9–6. Analysis & Synthesis Settings Note (1)		
Analysis & Synthesis Settings	Description	Value
STRATIX_OPTIMIZATION_TECHNIQUE	Type of optimization technique to use during Analysis & Synthesis stage of a Quartus II software compilation for a Stratix device	SPEED, AREA, BALANCED
CYCLONE_OPTIMIZATION_TECHNIQUE	Type of optimization technique to use during Analysis & Synthesis stage of a Quartus II software compilation for a Cyclone device	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 9–6:

- (1) Not all Analysis & Synthesis settings are available for all device families.

Table 9–7. Fitter Settings (Part 1 of 2) Note (1)		
Fitter Settings	Description	Value
AUTO_PACKED_REGISTERS_STRATIX	Register packing for Stratix devices	NORMAL, MINIMIZE_AREA, MINIMIZE_AREA_WITH_CHAINS
AUTO_PACKED_REG_CYCLONE	Register packing for Cyclone devices	OFF, MINIMIZE_AREA, MINIMIZE_AREA_WITH_CHAINS
INNER_NUM	PowerFit fitter effort level	{integer value}

Table 9–7. Fitter Settings (Part 2 of 2) Note (1)		
Fitter Settings	Description	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 9–7:

- (1) Not all Fitter settings are available for all device families.

Simple Custom Space

The custom space example below shows a simple custom exploration space that performs a seed sweep with various Analysis & Synthesis settings and Fitter settings.

```
<DESIGNSPACE>
  <POINT space="map">
  </POINT>
  <POINT space="fit">
  </POINT>
  <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 four points. The first two points in the space are special points: an empty “map” point and an empty “fit” point. DSE expects the first two points in any custom exploration space to be an empty map point and an empty fit point.

Following the empty map and fit points are one map point and one fit point that 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 is turned on; register packing is turned off; and a seed value of three is used.

Custom Space XML Schema

The following example contains an XML schema that describes 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.

```
<?xml version="1.0" encoding="UTF-8"?>
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
    <xss:element name="DESIGNSPACE">
        <xss:annotation>
            <xss:documentation>The root element of a design space
description</xss:documentation>
        </xss:annotation>
        <xss:complexType>
            <xss:sequence minOccurs="0" maxOccurs="unbounded">
                <xss:element ref="POINT"/>
            </xss:sequence>
            <xss:attribute name="project" type="xs:string" use="optional"/>
            <xss:attribute name="revision" type="xs:string" use="optional"/>
        </xss:complexType>
    </xss:element>
    <xss:element name="POINT">
        <xss:annotation>
            <xss:documentation>A point in the design space</xss:documentation>
        </xss:annotation>
        <xss:complexType>
            <xss:sequence minOccurs="0" maxOccurs="unbounded">
                <xss:element ref="PARAM"/>
            </xss:sequence>
            <xss:attribute name="space" type="xs:string" use="required"/>
            <xss:attribute name="enabled" type="xs:boolean" use="optional" default="1"/>
        </xss:complexType>
    </xss:element>
    <xss:element name="PARAM" type="xs:string" nillable="0">
        <xss:annotation>
            <xss:documentation>A single Quartus II software setting</xss:documentation>
        </xss:annotation>
    </xss:element>
</xss:schema>
```

Conclusion

DSE automates the process of finding the best set of options for your design. It explores the design space of your design by applying various optimization techniques and analyzing the results to shorten your design's timing closure cycle, achieve the smallest possible area, or reduce device utilization.

qii52009-2.2

Introduction

Available exclusively in the Altera® Quartus® II software, the LogicLock™ block-based design flow enables you to design, optimize, and lock down your design one module at a time. With the LogicLock methodology, 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. Additionally, you can reuse logic modules in other designs, further leveraging resources and shortening design cycles.

The Quartus II software beginning with version 4.2 supports the LogicLock block-based design flow for all of the following device families:

- Stratix® II, Stratix, Stratix GX, MAX® II, Cyclone™ II, Cyclone, APEX® and APEX II
- MAX II
- Excalibur™
- Mercury™ (Mercury devices support only locked and fixed regions)



This chapter assumes that you are familiar with the basic functionality of the Quartus II software. See the “LogicLock Module” in the Quartus II Online Tutorial for instructions on using the LogicLock feature in a sample design.

Improving Design Performance

You can use the LogicLock flow for performance optimization and preservation. You can use the LogicLock flow to place modules, entities, or any group of logic into regions in a device’s floorplan. Because LogicLock assignments are generally hierarchical, you have more control over the placement and performance of modules and groups of modules.

In addition to hierarchical blocks, you can apply LogicLock constraints to individual nodes, to make a wildcard path-based LogicLock assignment on a critical path, for example. 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.

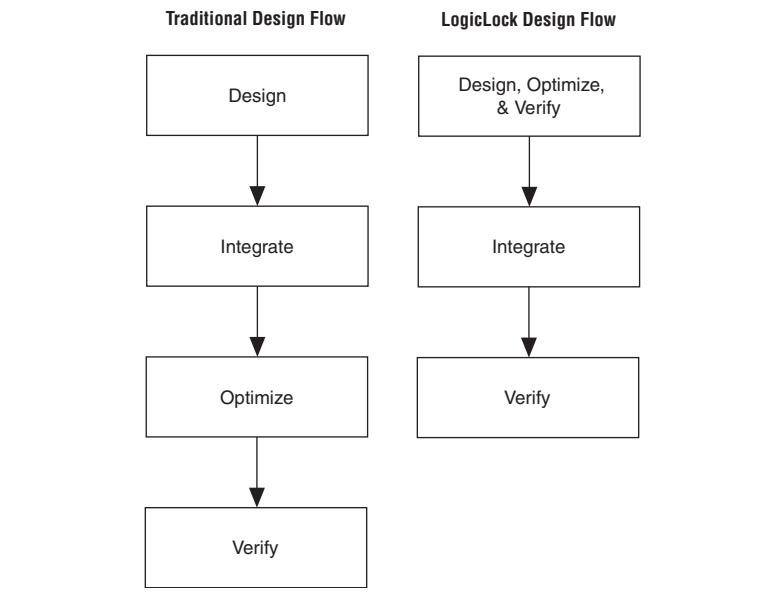
Block-Based Design with the Quartus II LogicLock Methodology

You can use the LogicLock design methodology in the Quartus II software to perform block-based hierarchical compilation. Using the LogicLock design flow, you can design and optimize each module independently, integrate all optimized modules into a top-level design, and verify the system. Incorporating each module into the top-level design does not affect the performance of the lower-level modules, as long as each module has registered inputs and outputs.

If each submodule in a design is represented by a unique netlist, only the portions of the design that have been updated must be resynthesized when you compile the design. You can make changes, optimize, and resynthesize the submodule you are working on without affecting other sections. Using the LogicLock design methodology, you can place the logic in each netlist file into 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.

[Figure 10–1](#) compares the traditional design flow with the LogicLock design flow.

Figure 10–1. Comparison of Traditional Design Flow with Quartus II LogicLock Design Flow



Preserving Timing Results Using the LogicLock Flow

When preserving logic placement in an Altera device, Altera recommends using an atom netlist to preserve the node names in sub-blocks of your design. An atom netlist contains design information that fully describes the submodule's 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 does change, 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.



Altera recommends that you 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** option to **Never Allow** for all nodes in the region, avoiding the possibility of a node name change in the top-level design when the region is recompiled.

You may need to remove previously back-annotated assignments for a modified block because the node names may be 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 the other areas of a design.

Designing with the LogicLock Feature

To design with the LogicLock feature, create a LogicLock region in a supported device and then assign logic to the region. The LogicLock region can contain any contiguous, rectangular block of device resources. After you have optimized the logic placed within the boundaries of a region to achieve the required performance, back-annotate the region's contents to lock the logic placement and routing. Then, when you integrate the region with the rest of the design, the performance is preserved.

This section explains the basics of designing with the LogicLock regions, including:

- Creating LogicLock regions
- Timing Closure Floorplan view
- LogicLock region properties
- Hierarchical (parent/child) LogicLock regions
- Assigning LogicLock region content
- Tcl scripts
- Quartus II block-based design flow
- Additional Quartus II LogicLock design features

Creating LogicLock Regions

There are four ways to create a LogicLock region:

- Using the **LogicLock Regions** window (Assignments menu)
- Using the **Create New Region** button in the Timing Closure Floorplan
- Using the **Hierarchy** tab of the **Project Navigator** (View menu)
- Using a Tool Command Language (Tcl) script

LogicLock Regions Window

The LogicLock window is comprised of the **LogicLock Regions** window and **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. You can modify a LogicLock region's size, state, width, height, and origin as well as whether the region is Soft or Reserved, in this window. When the region is back-annotated, the placement of the nodes within a region are relative to the region's origin, and the region's node placement during subsequent compilations is maintained.



For Stratix II, Stratix, Stratix GX, Cyclone II, Cyclone, and MAX II devices, the LogicLock region's origin is located at the bottom-left corner of the region. For all other supported devices, the origin is located at the top-left corner of the region.

The **LogicLock Regions** window displays any LogicLock regions that contain illegal assignments in red as shown in [Figure 10–2](#). If you make illegal assignments, you can use the **Repair Branch** command to reset the assignments for the currently selected region and its descendants to legal default values.



For more information on the **Repair Branch** command, see “[Repair Branch](#)” on page [10–23](#).

Figure 10–2. LogicLock Regions Window

Region name	Size	State	Width	Height	Origin	Soft region	Reserved
LogicLock Regions							
Root_region	Fixed	Locked	54	32	X0_Y0	Off	Off
<<new>>							
region_filter	Auto	Floating	1	1	<Illegal>	Off	Off
region_mult0	Auto	Floating	1	1	<Illegal>	Off	Off
region_mult1	Auto	Floating	1	1	<Illegal>	Off	Off
region_mult2	Auto	Floating	1	1	<Illegal>	Off	Off
region_mult3	Auto	Floating	1	1	<Illegal>	Off	Off

You can customize the **LogicLock Regions** window by dragging and dropping the various columns. The columns can also be hidden.



The Soft and Reserved columns are not shown by default.

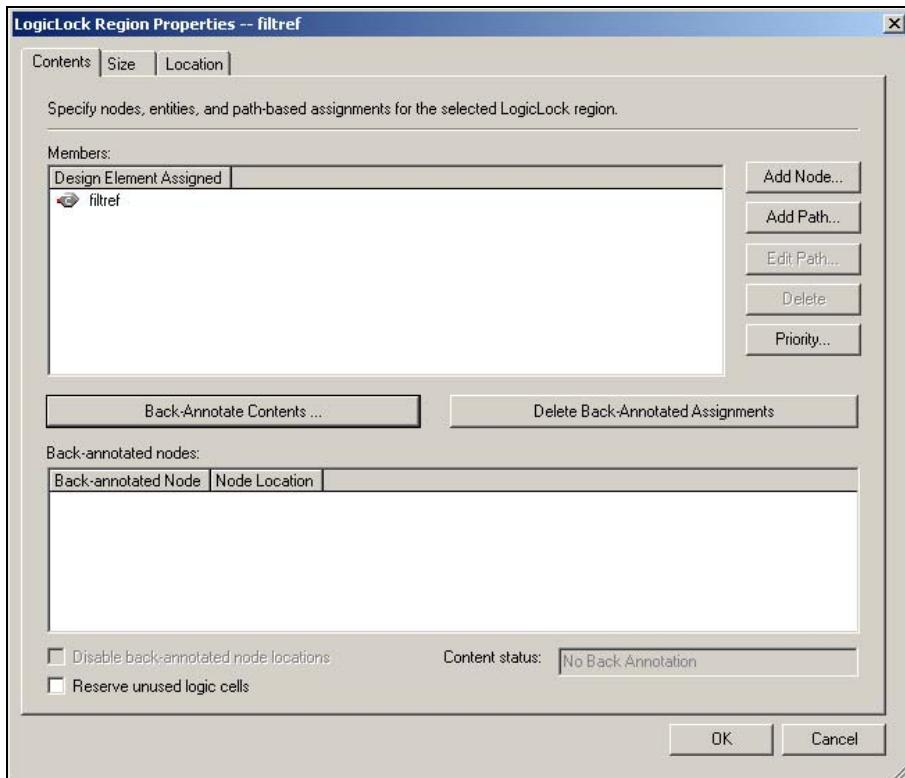
For designs targeting Stratix II, Stratix, Stratix GX, MAX II, Cyclone II, and Cyclone devices, the Quartus II software automatically creates a LogicLock region that encompasses the entire device. This default region is labelled **Root_region**, and it is effectively locked and fixed.

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 (see [Figure 10–3](#)). The **LogicLock Region Properties** dialog box shows the properties of the current selected regions.



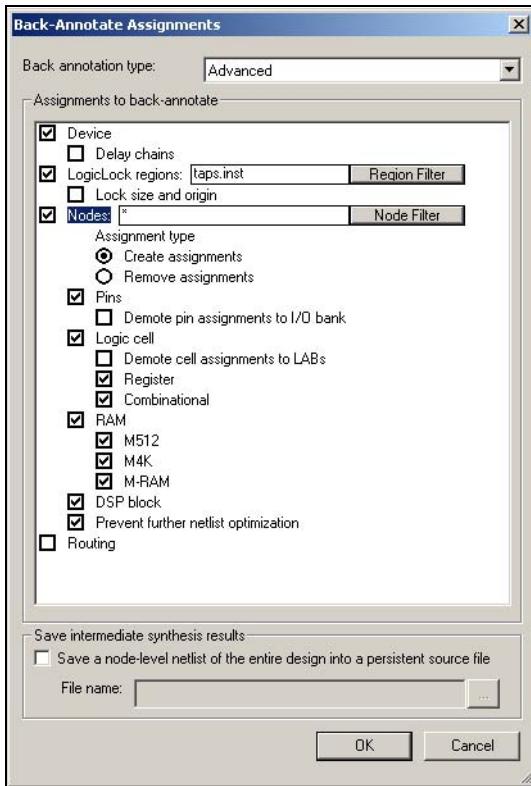
The **LogicLock Region Properties** dialog box can be opened by double-clicking any region in the **LogicLock Regions** window or right-clicking the region and selecting **Properties**.

Figure 10–3. LogicLock Region Properties Dialog Box



To back-annotate the contents of your LogicLock regions, perform these steps:

1. In the **LogicLock Region Properties** dialog box, click **Back-Annotate Contents**.
2. Select the contents you wish to back-annotate using the **Back-Annotate Assignments (Advanced type)** (Assignments menu) dialog box (Figure 10–4)
3. Click **OK**.

Figure 10–4. Back-Annotate Assignments Dialog Box (Advanced Type)

You can also back-annotate routing within LogicLock regions for increased region portability. For more information on back-annotating routing information, see “[Back-Annotating Routing Information](#)” on page 10–34.

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 (e.g., logic array block [LAB], M512, M4K, M-RAM, digital signal processing [DSP] block, etc.) under **Node Location**. Each node’s location is the placement of the node after the last 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, then the nodes are assigned to LABs rather than directly to logic cells.

Timing Closure Floorplan Editor

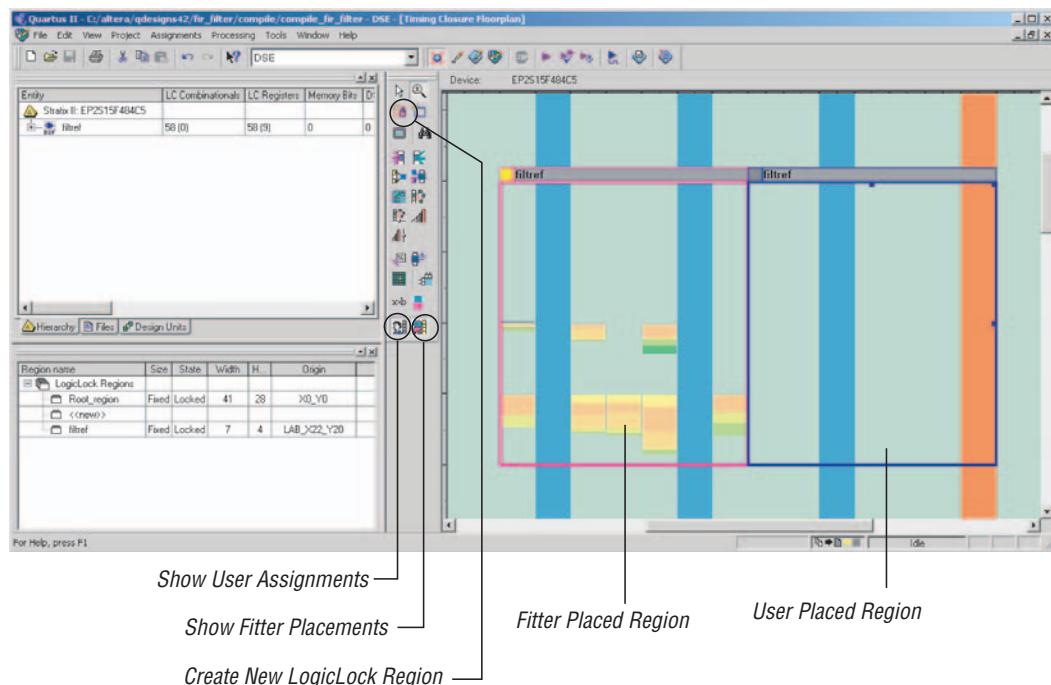
The Timing Closure Floorplan Editor has toolbar buttons that are used to manipulate LogicLock regions, as shown in [Figure 10–5](#). You can use the **Create New LogicLock Region** button to draw LogicLock regions in the device floorplan.



The Timing Closure Floorplan Editor displays LogicLock regions when **Show User Assignments** or **Show Fitter Placements** is selected. 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 enabled in the Timing Closure Floorplan, current assignments made to a LogicLock region are visible. When the **Fitter Placement** option is enabled, you can see the properties of the LogicLock region after the last compilation. User-assigned LogicLock regions appear in the Floorplan Editor with a dark blue border ([Figure 10–5](#)). Fitter-placed regions appear in the Floorplan Editor with a magenta border.

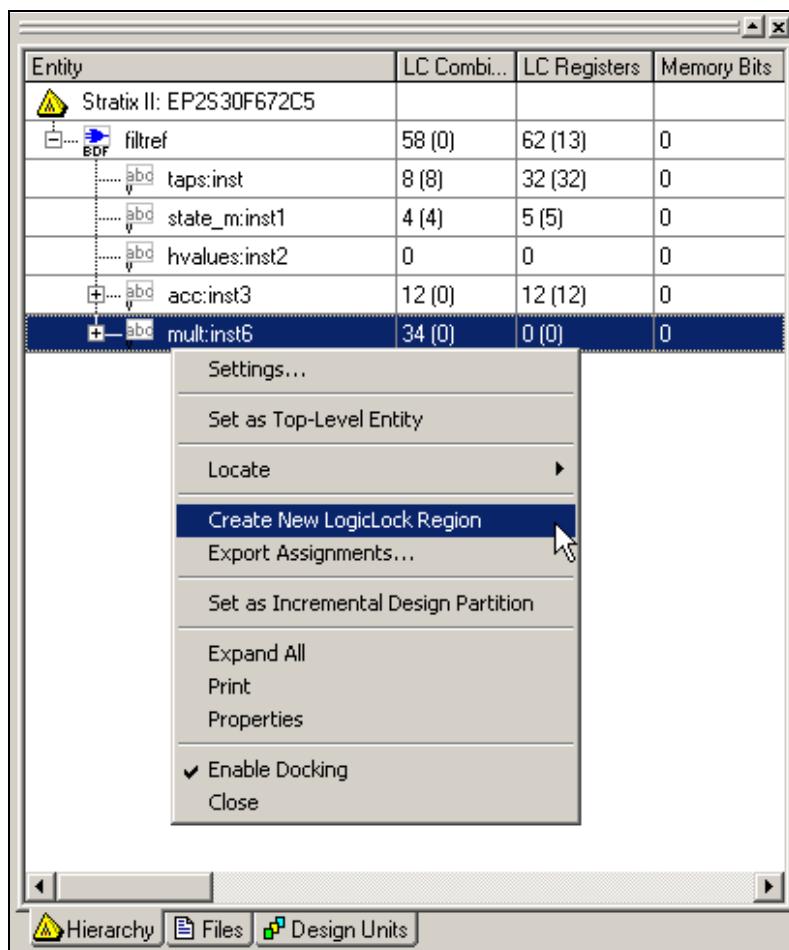
Figure 10–5. Floorplan Editor Toolbar Buttons



Design Hierarchy

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 **Hierarchy** tab of the **Project Navigator** (View menu). With the hierarchy of the design fully expanded, as shown in [Figure 10–6](#), you can conveniently create a LogicLock region by right clicking on any design entity in the design and selecting **Create New LogicLock Region** in the right button pop-up menu.

Figure 10–6. Project Navigator Used to Create LogicLock Regions



Tcl Scripts

You can create LogicLock regions and assign nodes to them with Tcl commands that you can run from the Tcl Console or at the command prompt.

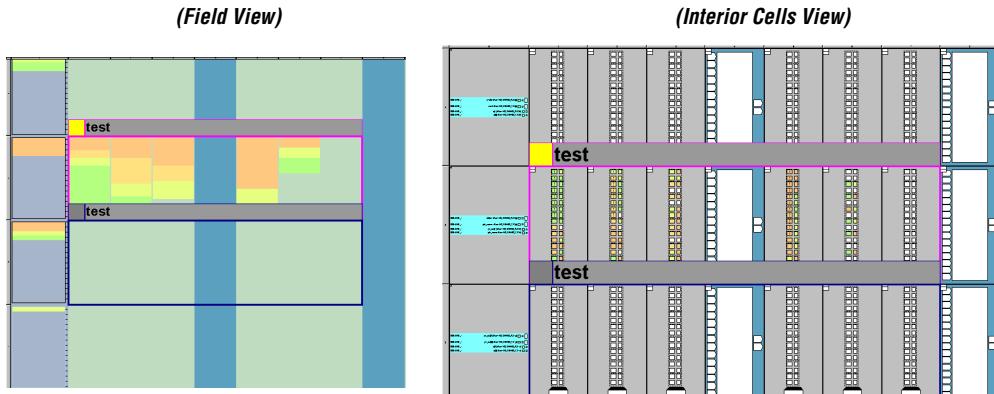


For more information, refer to the *Scripting Support* chapter in Volume 2 of the *Quartus II Handbook*.

Timing Closure Floorplan View

The **Timing Closure Floorplan** view provides you with current and last 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 10–7](#). 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 10–7. Timing Closure Floorplan Editor



LogicLock Region Properties

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 and/or location of a region, or the Quartus II software can generate them automatically. The Quartus II software bases the size and location of the region on its contents and the module's timing requirements. [Table 10–1](#) describes the options for creating LogicLock regions.

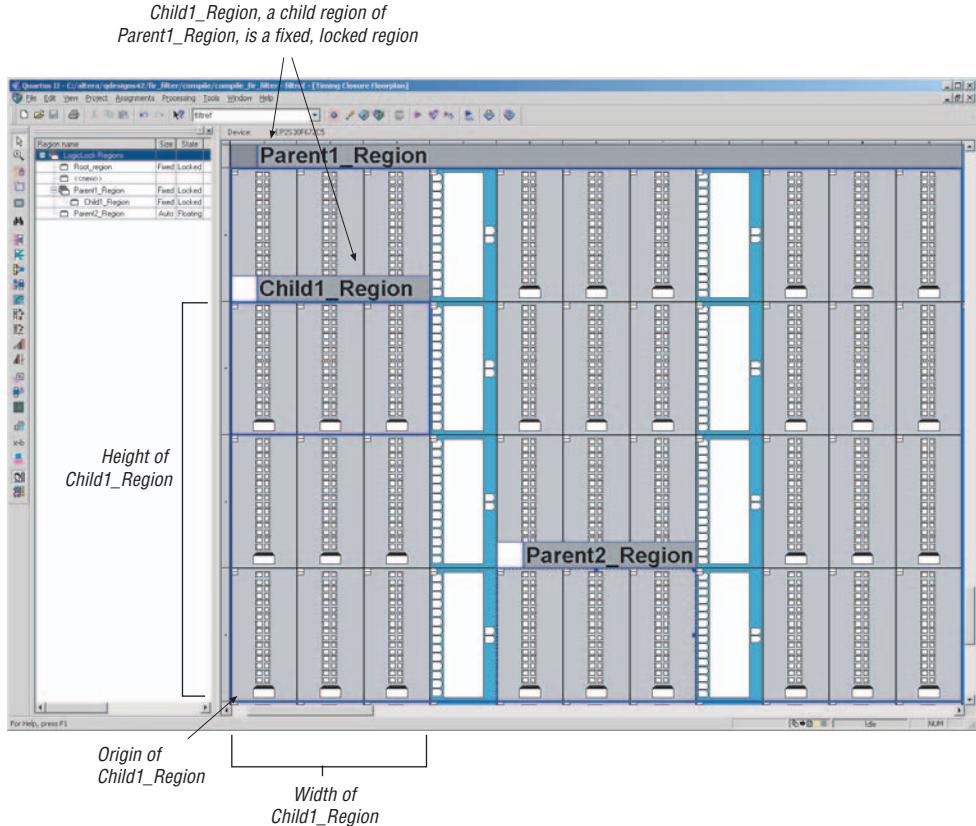
Table 10–1. Types of LogicLock Regions

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.
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 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 Stratix, Stratix II, Stratix GX, MAX II, Cyclone II, and Cyclone the origin is located in the lower-left hand corner. The origin is located in the upper-left corner for other families.



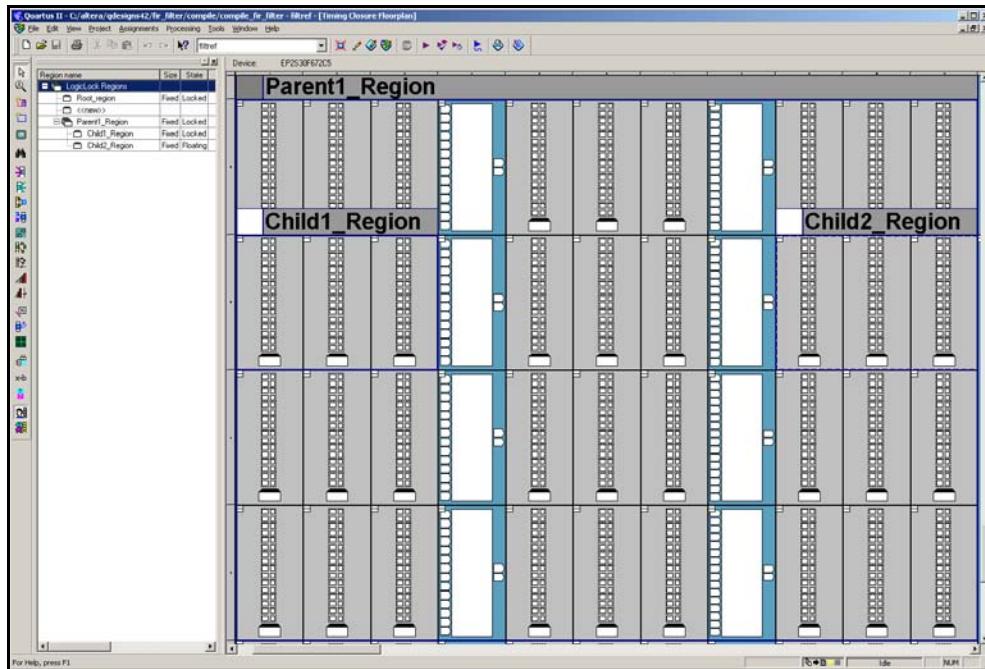
The Quartus II software cannot automatically define a region's size if the location is locked. Therefore, if you want to specify the exact location of the region, you must also specify the size. Mercury devices support only locked and fixed regions.

The floorplan excerpt in [Figure 10–8](#) shows the LogicLock region properties for a design implemented in a Stratix device.

Figure 10–8. LogicLock Region Properties

Hierarchical (Parent and/or Child) LogicLock Regions

With the LogicLock design flow, you can define a hierarchy for a group of regions by declaring parent and/or 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. [Figure 10–9](#) shows an example child region within a parent region, including labels for a locked location and floating location in a Stratix II device.

Figure 10–9. Child Region within a 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, while 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, it must fit entirely within the parent region.

Assigning LogicLock Region Content

Once you have defined a LogicLock region, you must assign resources to it using the **Timing Closure Floorplan**, the **LogicLock Regions** dialog box, the Assignment Editor, or Tcl scripts with the Quartus II Tcl Console or the **quartus_sh** executable.

Using Drag & Drop to Place Logic

You can drag selected logic from the **Hierarchy** tab of the **Project Navigator**, Node Finder, or a schematic design file and drop it into the Timing Closure Floorplan or the **LogicLock Regions** dialog box.

Figure 10–10 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.

Figure 10–10. Drag & Drop Logic in the Timing Closure Floorplan

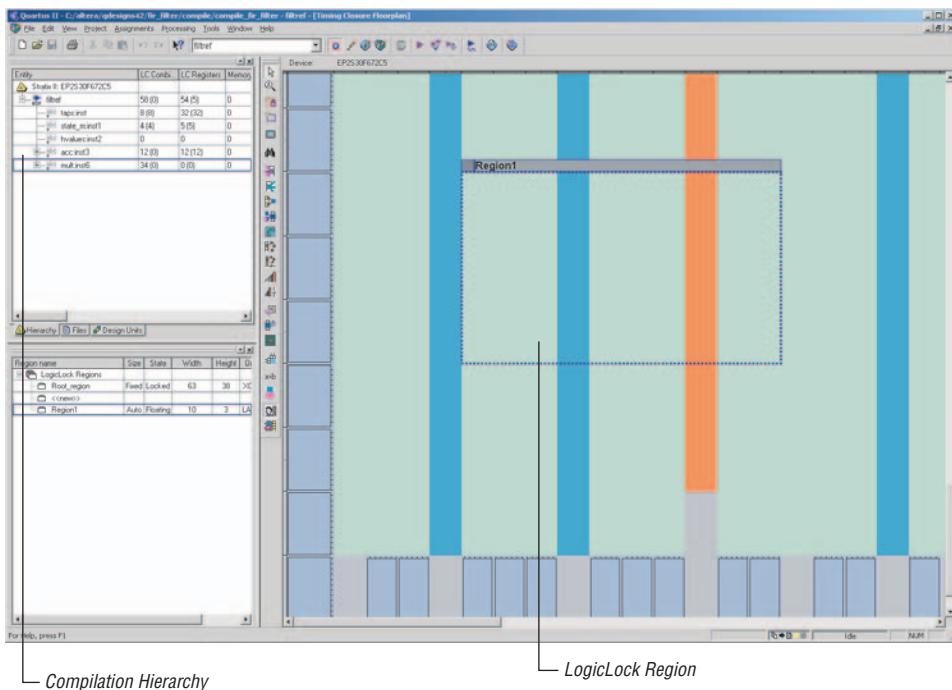
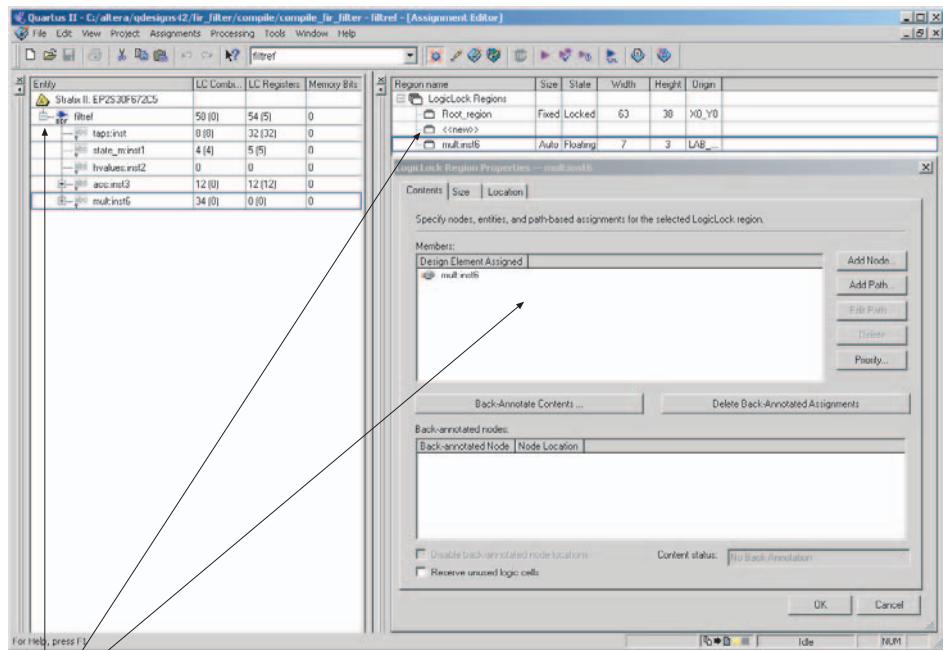


Figure 10–11 shows logic that has been dragged from the **Hierarchy** tab of the **Project Navigator** and dropped into the **LogicLock Regions Properties** dialog box. Logic can also be dropped into the **Design Element Assigned** column of the **Contents** tab of the **LogicLock Region Properties** box.

Figure 10–11. Drag & Drop Logic into the LogicLock Regions Dialog Box

Drag and drop from the Hierarchy tab of the Project Navigator to the LogicLock Regions window or into the Design Element Assigned column under Members in the LogicLock Region Properties dialog box.



You must manually assign pins to a LogicLock region. 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 Stratix, Stratix II, MAX II, Cyclone II, and Cyclone devices, the locked regions must include the I/O pins as resources.

Using the Assignment Editor to Place Logic

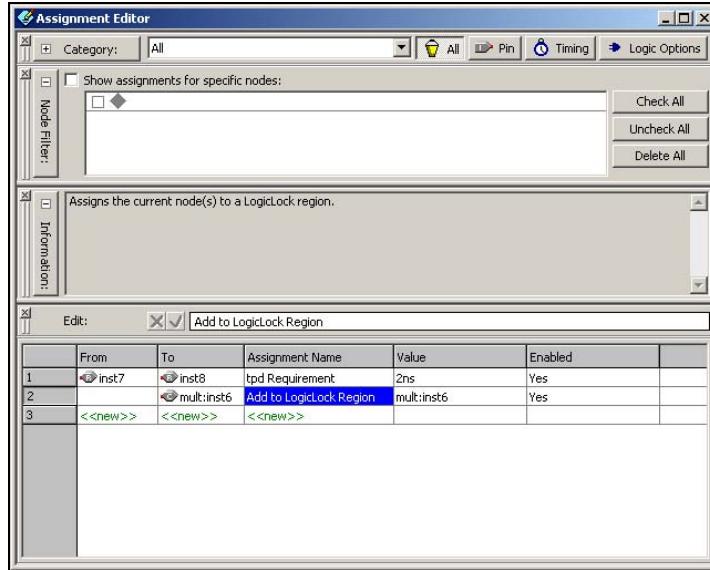
You can also use the Assignment Editor to assign entities and nodes to a LogicLock region (see [Figure 10–12](#)). To assign content to a LogicLock region with the Assignment Editor, perform the following steps:

1. Under **Assignment Name**, select **Add to LogicLock Region**.
2. Under **Value**, specify your LogicLock region name.

3. Under **To**, specify either nodes or entities that are to reside in the LogicLock region.

The nodes or entities are then assigned to the selected LogicLock region.

Figure 10–12. Assignment Editor



Tcl Scripts

You can create LogicLock regions and assign nodes to them with Tcl commands that you can run from the Tcl Console or at the command prompt. The Tcl command `set_logiclock` is used to create or change the attributes of LogicLock regions.



For more information on creating and using LogicLock regions and contents, see the *Command Line* and *Tcl API* topics in the Quartus II online Help or “[Scripting Support](#)” on page 10–37.

Quartus II Block-Based Design Flow

When using the LogicLock design flow, divide the design into modules and perform the following steps in the Quartus II software for each module:

1. Synthesize the module using the Quartus II software or another synthesis tool.
2. Optimize the module in the Quartus II software.
3. Export the module and the LogicLock constraints.
4. Import all modules and LogicLock constraints into the top-level project.
5. Compile and verify the top-level design.

Synthesize the Module

You can synthesize the module in the Quartus II software or any Altera-supported third-party synthesis tool, e.g., the Synplify[®], LeonardoSpectrumTM, or FPGA Compiler II software. The software synthesizes each module and creates an atom netlist, which represents the logic in terms of Altera primitives for the target device.

In the atom netlist, the nodes are fixed as Altera primitives; the node names do not change if the atom netlist does not change. If a node name does change, any placement information made to that node is invalid and ignored. Third-party tools generate atom netlists as EDIF Input Files (.edf) or Verilog Quartus Mapping Files (.vqm).

Optimize the Module

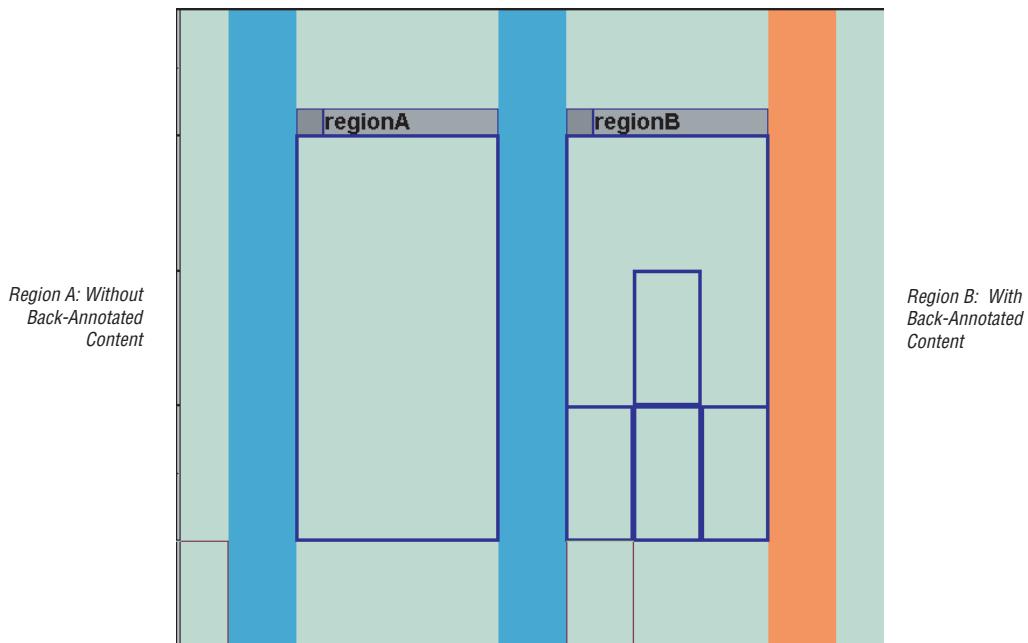
Before optimizing a module in the Quartus II software, create a project with the module as the top-level entity. You must assign the module to a single (or multiple) LogicLock region. See “[Constraint Priority](#) on [page 10-31](#) for information on the precedence of the LogicLock region and other constraint settings.

After you have optimized the module so that it meets your timing requirements, lock down the placement of nodes in a LogicLock region by back-annotating the contents of the region. To make relative location assignments, you must fix the node names. Fixed node names require an atom netlist so that the assignments for each node remain valid. The node placement is fixed relative to the LogicLock region for the module.

For the Quartus II software to achieve optimal placement, you should make timing assignments for all clock signals in the design, including t_{SU} , t_{CO} , and t_{PD} .

To facilitate the LogicLock design flow, the **Timing Closure Floorplan** highlights resources that have back-annotated LogicLock regions. [Figure 10–13](#) shows a back-annotated LogicLock region in the **Timing Closure Floorplan**.

Figure 10–13. Back-Annotated LogicLock Region



Export the Module

This section describes how to export a module's constraints to a format that can be imported by a top-level design. To be exported, a module requires design information as an atom netlist (VQM or EDF), placement information stored in a Quartus Settings File (.qsf), and routing information stored in a Routing Constraint 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 need to generate another atom netlist. However, if you use any Synthesis Netlist Optimizations or Physical Synthesis Optimizations you must generate a Quartus II VQM, 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 as a VHDL Design File (.vhd), Verilog Design File (.v), Text Design File (.tdf), or a 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 VQM File in the **atom_netlists** subdirectory in the project directory.



If the atom netlist is from a third-party synthesis tools and the design has a black-boxed library of parameterized modules (LPM) functions or Altera megafunctions, you must generate a Quartus II VQM File for the black-boxed modules.



For instructions on creating an atom netlist in the Quartus II software, see *Saving Synthesis Results for an Entity to a Verilog Quartus Mapping File* in Quartus II Help.

When you export LogicLock regions, the Quartus II software defaults to exporting your entire design's LogicLock region assignments. However, you can export a sub-entity of the compilation hierarchy and all of its relevant regions. This can be accomplished by right-clicking the entity in the **Hierarchy** tab of the **Project Navigator** and selecting **Export Assignments** from the right button pop-up menu.

Placement Information

The QSF 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 RCF contains the module's LogicLock routing information. To maintain performance, you must back-annotate the module.



For instructions on exporting a LogicLock region assignment in the Quartus II software, see *Exporting LogicLock Region Assignments & Other Entity Assignments* in Quartus II Help.

Import the Module

You can specify which QSF is used for a specific instance or entity with the **LogicLock Import File Name** option in the Assignment Editor. Therefore, you can 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 QSF and RCF 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, thus ensuring that the Quartus II software is aware of all instances of the lower-level modules.

The following sections describe how to specify a QSF for a module and how to import the LogicLock assignments into the top-level design.

Specify the QSF and Atom Netlist

To specify the QSF and atom netlist to import, perform the following steps:

1. Specify an atom netlist for the module that you are importing by either copying the atom netlist to your current working directory or choosing **Add/Remove Project Files** (Project menu) and browsing to the file.
2. Perform an analysis and elaboration.
3. Expand the design hierarchy on the **Hierarchy** tab of the **Project Navigator** by clicking the + icon next to the top-level entity.
4. Right-click on the entity and choose **Locate** in the Assignment Editor.

5. Under **Assignment Name**, choose **LogicLock Import File Name**.
6. Under **Value**, type the name and relative path to the QSF, or click **Browse** and navigate to the QSF in the **Select File** dialog box.

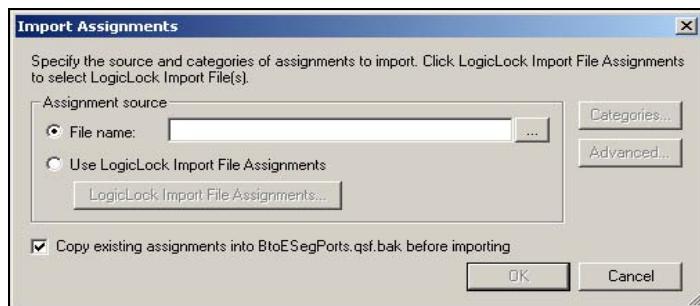
Repeat steps 3 through 5 for all instances of all entities that require a specific QSF.

You can follow the same procedure for specifying a QSF when specifying an RCF. Instead of selecting **LogicLock Import File Name**, select **LogicLock Back Routing Constraints File Name**.

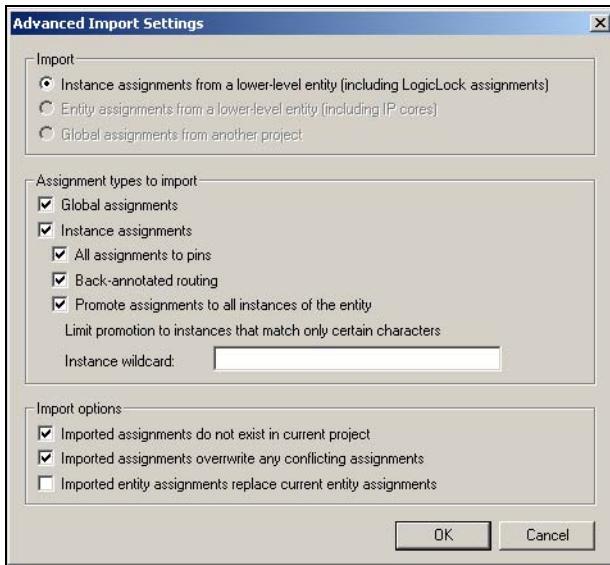
Import the Assignments

To import the assignments, choose **Import Assignments** (Assignments menu). [Figure 10–14](#) shows the **Import Assignments** dialog box.

Figure 10–14. Import Assignments Dialog Box



There are several options available in the **Advanced Import Settings** dialog box that you can use to control the importing of your LogicLock regions, as shown in [Figure 10–15](#).

Figure 10–15. Advanced Import Settings Dialog Box

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 will remain as specified in the QSF. This 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 & 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 have been met.

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 Hierarchy tab of the **Project Navigator** or **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, select **Show Intra-region Delay** (View menu).

Repair Branch

When you retarget your design to either a larger or smaller device, there is a chance that your LogicLock regions no longer contain valid values for location or size in the new device, resulting in an illegal LogicLock region. The Quartus II software identifies illegal LogicLock regions in the **LogicLock Regions** dialog box by coloring the name of the region containing the error red.

To correct the illegal LogicLock region, use the **Repair Branch** command. Right click the desired LogicLock region's name and select **Repair Branch** (right button pop-up menu).

If more than one illegal LogicLock region exists, you can repair all regions by right clicking the first line in the **LogicLock** window that contains the text **LogicLock Regions** and selecting **Repair Branch**.

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 **Reserve unused logic cells** on the **Contents** 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 device floorplanning. 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 entering into 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 sub-entity 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.

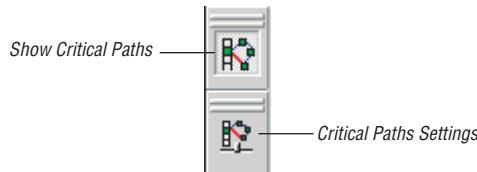
Rubber Banding

Choose **Routing > Rubber Banding** (View menu) 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 within a LogicLock region by turning the **Show Critical Paths** option On. This option is used in conjunction with the **Critical Paths Settings** option that allows you to display paths based on the Timing Analysis report, as shown in [Figure 10–16](#).

Figure 10–16. Show Critical Paths & Critical Paths Settings



Show Connection Count

You can determine the number of connections between LogicLock regions by turning the **Show Connection Count** option On.

Analysis & Synthesis Resource Utilization by Entity

The Compilation Report contains an **Analysis & 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.

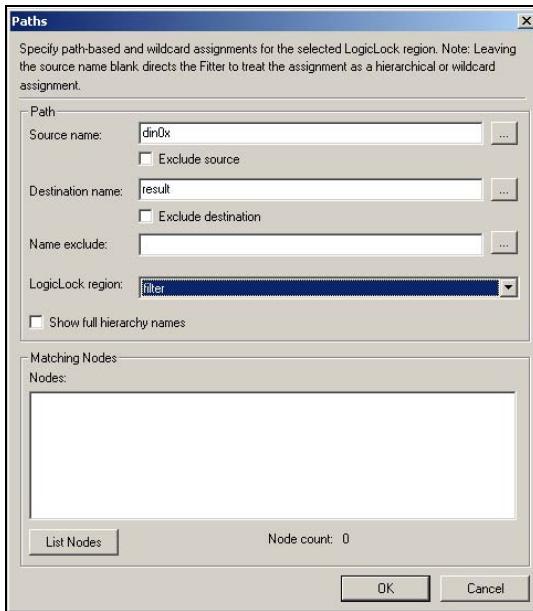
Path-Based Assignments

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. The path's source and destination nodes must be a valid register-to-register path, meaning that the source and destination nodes must be registers. [Figure 10–17](#) shows the **Paths** dialog box.

The **Paths** dialog box is opened by clicking **Add Path** or **Edit Path** on the **Contents** tab of the **Logic Lock Regions Dialog Box**.



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.

Figure 10–17. Paths Dialog Box

You can also use the Quartus II Timing Analysis Report to create path-based assignments. To create path-based assignments, follow these steps:

1. Expand the Timing Analyzer section in the Compilation Report.
2. Select any of the clocks in the section that is labelled “Clock Setup:<clock name>”
3. Locate a path that you would like to assign to a LogicLock region. Drag this path from the Report window and drop it in the <<new>> section of the LogicLock Region window.

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 provides the Revisions feature that allows for a convenient way to organize the same project with different settings until an optimum configuration is found.

Use the **Revisions** dialog box (Project menu) to create and set revisions. Revision can be based on the current design or any previously created revisions. A description can also be entered for each revision created. This is a convenient way to organize the placement constraints created for your LogicLock regions.

LogicLock Assignment Precedence

Conflicts might 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 Quartus II software's order of precedence is as follows from highest to lowest:

1. Exact node-level assignments
2. Path-based and wildcard assignments
3. Hierarchical assignments

However, conflicts might 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 keeps the order in which the assignments were made and treats the last assignment created with the highest priority.



Open the **Priority** dialog box by selecting **Priority** on the **Contents** tab of the **LogicLock properties** dialog box. You can change the priority of path-based and wildcard assignments by using the **Up** or **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 Vs Soft LogicLock Regions

Normally all nodes assigned to a particular LogicLock region always 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 a LogicLock region as being “Soft,” the Quartus II software attempts to place as many nodes assigned to the region as close together as possible, and has the added flexibility of moving nodes outside of the soft region to meet your design’s performance requirement. This 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 will not be 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, Stratix GX, Stratix II, MAX II, Cyclone II, or Cyclone 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.

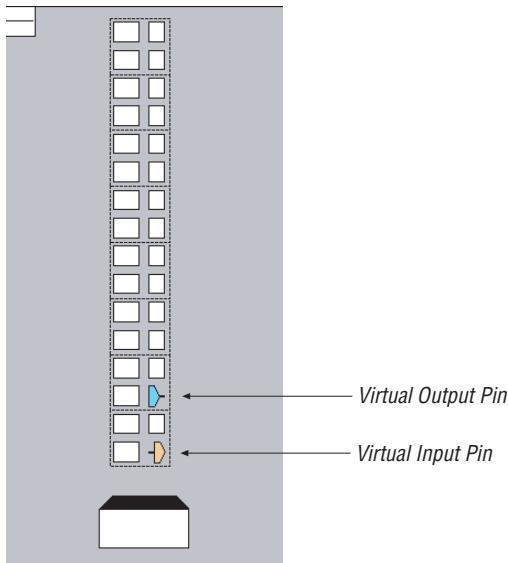
Soft LogicLock regions cannot be back-annotated because the Quartus II software may have placed nodes outside of the LogicLock region resulting in undefinable location assignments relative to the region’s origin and size.

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. This I/O port mapping may create problems for a modular/hierarchical design because lower-level modules may have more I/O ports than pins available on the targeted device, or the I/O ports may not directly feed a device pin, but may drive other internal nodes. The Quartus II software supports virtual pins to accommodate this situation. Virtual pin assignments tell the Quartus II software which I/O ports of the design module become internal nodes in the top-level design. These assignments prevent the number of I/O ports in the lower-level module from exceeding the total number of available device pins. Every I/O port that is designated as a virtual pin gets mapped to an LCELL register in the device. [Figure 10–18](#) shows the virtual input and output pins in the **Floorplan Editor**.

Figure 10–18. Virtual I/O Pins in the Quartus II Floorplan Editor



Bidirectional, registered I/O pins, and I/O pins with output enable signals cannot be virtual pins. All virtual pins must map to device I/O pins in the top-level design.

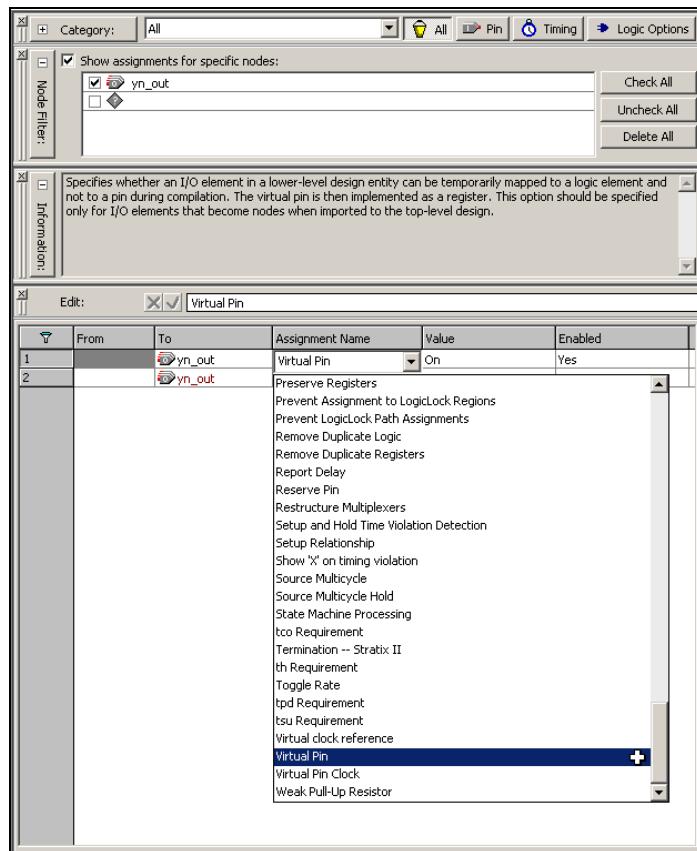
In the top-level design, these virtual pins are connected to an internal node in another module. Making assignments to virtual pins allows you to place them in the same location or region on the device as the

corresponding internal nodes would exist in the top-level module. This feature has the added benefit of providing accurate timing information during lower-level module optimization.

To accommodate designs with multiple clock domains, you can specify individual clock signals by turning on the **Virtual Pin Clock** option on for each virtual pin.

 Virtual pin and virtual pin clock assignments are made with the **Assignment Editor**. Figure 10–19 shows assigning virtual pins using the **Assignment Editor**.

Figure 10–19. Using the Assignment Editor to Assign Virtual Pins



 Setting **Filter Type** to **Pins: Virtual** allows the Node Finder to display all assigned virtual pins in the design.

LogicLock Restrictions

This section discusses restrictions that you should consider when using the LogicLock design flow, including:

- Constraint priority
- Placing LogicLock regions
- Placing memory, pins, and other device features into LogicLock regions

Constraint Priority

During the design process, it is often necessary to place restrictions on nodes or entities in the design. Often, these restrictions conflict with the node or entity assignments for a LogicLock region. To avoid conflicts, you should 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.

Placing LogicLock Regions

A fixed region must contain all of the 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:

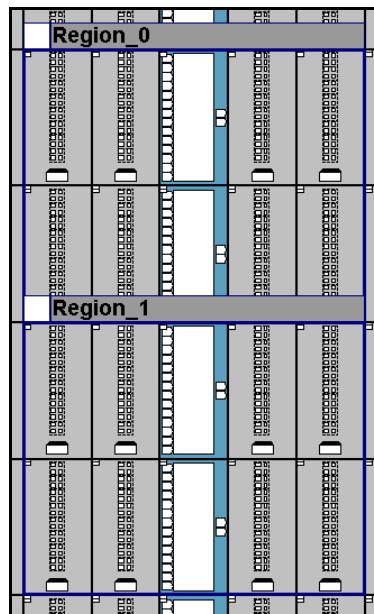
- LogicLock regions with pin assignments must be placed on the periphery of the device, adjacent to the pins. (For Stratix, Stratix GX, Stratix II, MAX II, Cyclone II, and Cyclone devices, you must also include the I/O block.)
- Floating LogicLock regions cannot overlap.
- Avoid creating fixed and locked regions that overlap.
- After back-annotating a region, the software can place the region only in areas on the device with exactly the same resources.



These guidelines are particularly important if you want to import multiple instances of a module into a top-level design, because 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.

Figure 10–20 shows a floorplan with two instantiations of the same module. Both modules have the same LogicLock constraints and require exactly the same resources. The Quartus II software places the two LogicLock regions in different areas of the device that have the same resources.

Figure 10–20. Floorplan of Two Instances of a LogicLock Region



Placing Memory, Pins & Other Device Features into LogicLock Regions

A LogicLock region includes all device resources within its boundaries. 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.

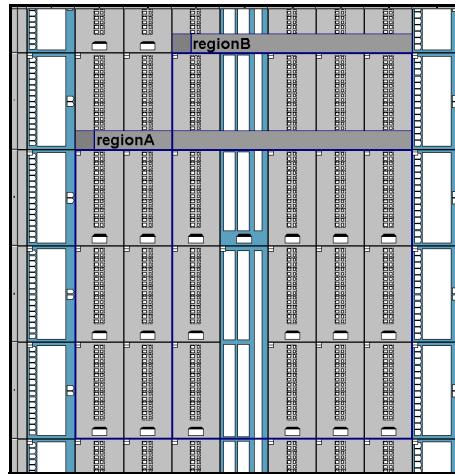


Pin assignments to LogicLock regions are effective only in fixed and locked regions. Pins assigned to floating regions do not influence the region's placement.

Only one LogicLock region can claim a device resource. If the boundary includes part of a device resource, such as a DSP block, the Quartus II software allocates the entire resource to the LogicLock region.

Figure 10-21 shows two overlapping regions in the same Stratix DSP block. The Quartus II software can assign this resource to only one of the LogicLock regions. The region's resource requirements determine which region gets the assignment. If both regions require a DSP block, the Quartus II software issues a fitting error.

Figure 10-21. Overlapping LogicLock Regions



Back-Annotating Routing Information

LogicLock regions not only allow you to preserve the placement of logic from one compilation to the next, but 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 if necessary because this can prevent the Quartus II Fitter from finding an optimal fit for your design.

You can back-annotate the routing by selecting **Routing** in the **Back-Annotate Assignments** dialog box (Assignments menu). See [Figure 10–4 on page 10–7](#).



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 in the **Back-Annotate Assignments** (Assignments menu) dialog box if back-annotation of routing is selected. Writing out a VQM file causes the Quartus II software to enforce persistent naming of nodes when saving the routing information. The VQM 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.

Exporting Back-Annotated Routing in LogicLock Regions

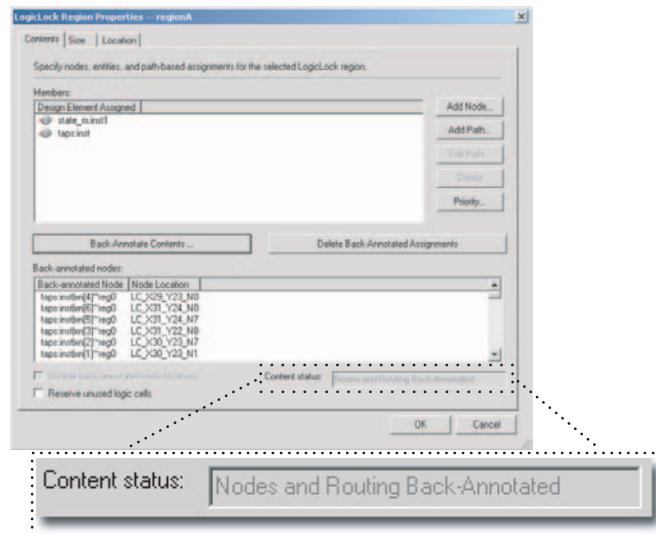
You can export the LogicLock region routing information by turning on the **Export Back-annotated routing** option on in the **Export Assignments** dialog box (Assignments menu). This generates a QSF and a RCF in the specified directory. The QSF contains all LogicLock region properties as specified in the current design. The RCF contains all the necessary routing information for the exported LogicLock regions.

This RCF 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.

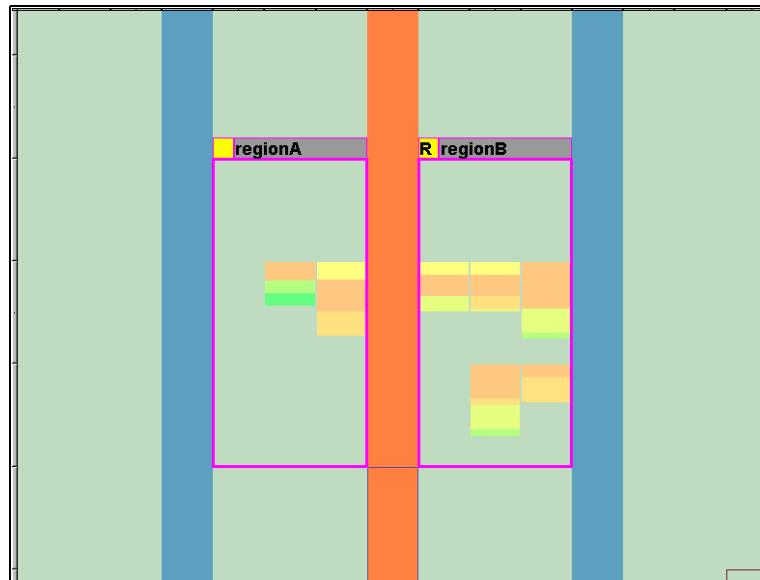
To determine if a LogicLock region contains back-annotated routing, see the **Content Status** box shown on the **Contents** tab of the **LogicLock Region Properties** dialog box. If routing has been back-annotated, the status is “Nodes and Routing Back-Annotated,” as shown in [Figure 10–22](#).

Figure 10–22. LogicLock Status



The Quartus II software also reports whether routing information has been back-annotated in the **Timing Closure Floorplan Assignments** menu). LogicLock regions with back-annotated routing have an “R” in the top-left hand corner of the region, as shown in [Figure 10–23](#).

Figure 10–23. Back-Annotation of Routing

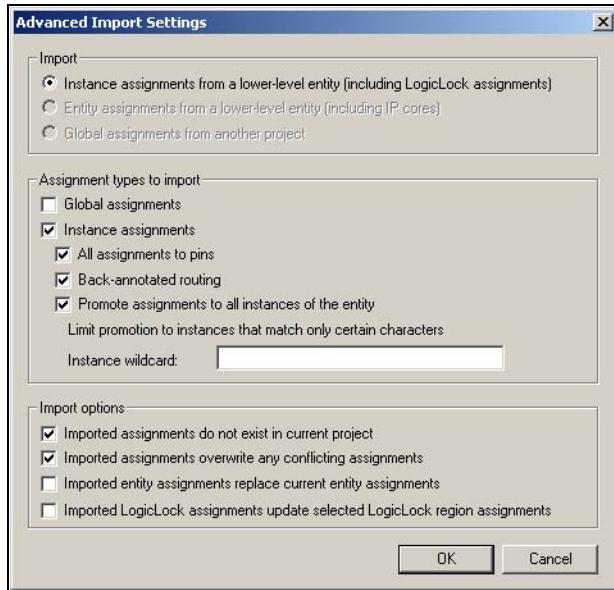


Importing Back-Annotated Routing in LogicLock Regions

To import LogicLock region routing information, turn on the **Back-annotated routing** option in the **Advanced Import Settings** dialog box. Access the **Advanced Import Settings** dialog box by clicking **Advanced** in the **Import Settings** dialog box (Assignments menu). [Figure 10–24](#) shows this dialog box. The Quartus II software imports and applies all LogicLock region assignments for the appropriate instances automatically.



An RCF must be explicitly specified using the LogicLock **Back-annotated Routing Import File Name** option for the Quartus II software to import routing information for your design.

Figure 10–24. Import LogicLock Regions

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 RCF for a lower-level entity, you must use the same atom netlist, that is, the VQM that was used to generate the RCF. This ensures that the node names annotated in the RCF match those in the atom netlist.

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 ↵
```



For more information about Tcl scripting, see the *Tcl Scripting* chapter in Volume 2 of the *Quartus II Handbook*. For more information about command-line scripting, see the *Command-Line Scripting* chapter in Volume 2 of the *Quartus II Handbook*.

Initializing & 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 uninitialized 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 will be 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, see “[Creating LogicLock Regions](#)” on page 10–4.

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 10–13.

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 into a Persistent Source File (.vqm)

Make the following assignments to cause the Quartus II Fitter to save a node-level netlist into a VQM 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 must be 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 VQM file is saved in the directory specified at the completion of a full compilation.



For more information about saving a node-level netlist, see “[Atom Netlist Design Information](#)” on page 10–19.

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 see “[Export the Module](#)” on page 10–18.

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, see “[Import the Module](#)” on page 10–20.

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]
foreach member [get_logiclock_member_priority] {
    set reverse [insert $reverse 0 $member]
}
set_logicclock_member_priority $reverse
```



For more information about setting the LogicLock assignment priority, see “[Constraint Priority](#)” on page 10–31.

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, see “[Virtual Pins](#)” on page 10–29.

Back-Annotating LogicLock Regions

Use the following command line option to back-annotate a design called `my_project` and demote assignments to LAB-level assignments.

```
quartus_cdb --back_annotate=lab my_project
```



For more information about Tcl scripting, see the *Tcl Scripting* chapter in Volume 2 of the *Quartus II Handbook*. For more information about command-line scripting, see the *Command-Line Scripting* chapter in Volume 2 of the *Quartus II Handbook*.

Conclusion

The LogicLock block-based design flow shortens design cycles because it allows design and implementation of design modules to occur independently, and preserves performance of each design module during system integration. You can export modules, making design reuse easier.

You can 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 so that you can use LogicLock region assignments to improve the performance of non-hierarchical designs.

qii52011-1.0

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, see the *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 4.0
- Amplify Physical Optimizer, version 3.2

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.



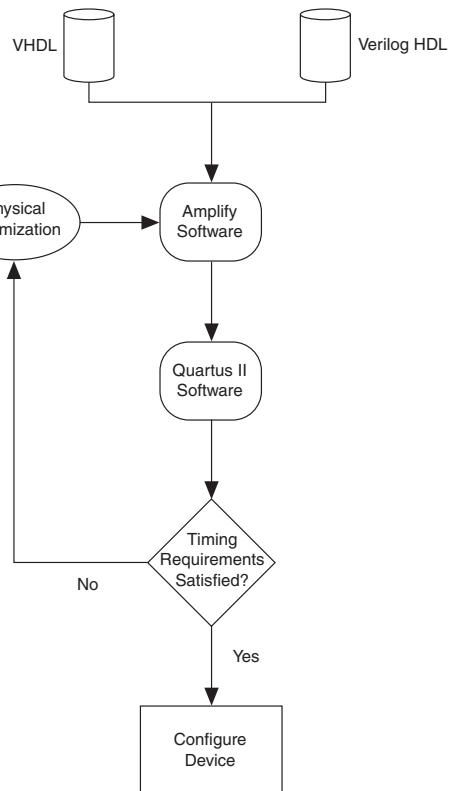
See “[Forward Annotating Amplify Physical Optimizer Constraints into the Quartus II Software](#)” on page 11–12 for more information on setting up a Quartus II project with Amplify Physical Optimizer Tcl script files.

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 *<project 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 11–1](#) shows the Amplify Physical Optimizer design flow.

Figure 11–1. Software Design Flow



Initial Pass: No Physical Constraints

The initial iteration involves synthesizing the design in the Amplify Physical Optimizer software without physical constraints.

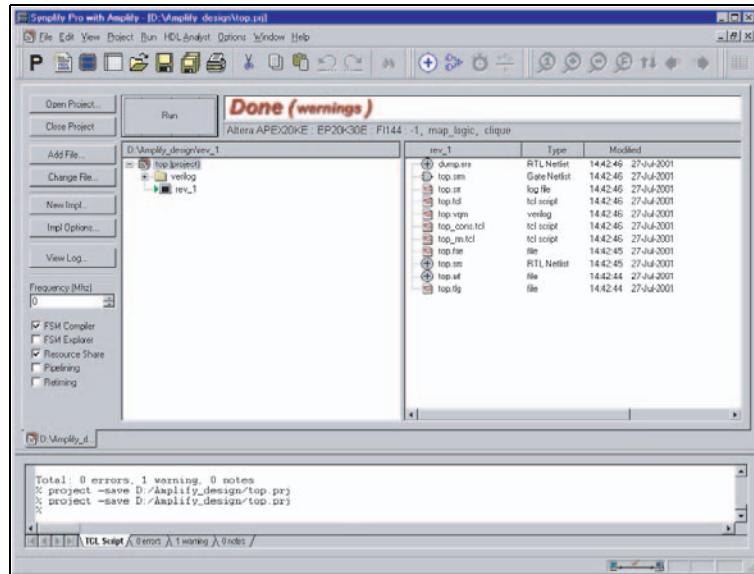
Before beginning the physical synthesis flow, run an initial pass in the Amplify Physical Optimizer without physical constraints. At the completion of every Quartus II compilation, the Quartus II Timing Analyzer performs a comprehensive static timing analysis on your design and reports your design's 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 11–2](#) 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 11–2. 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.

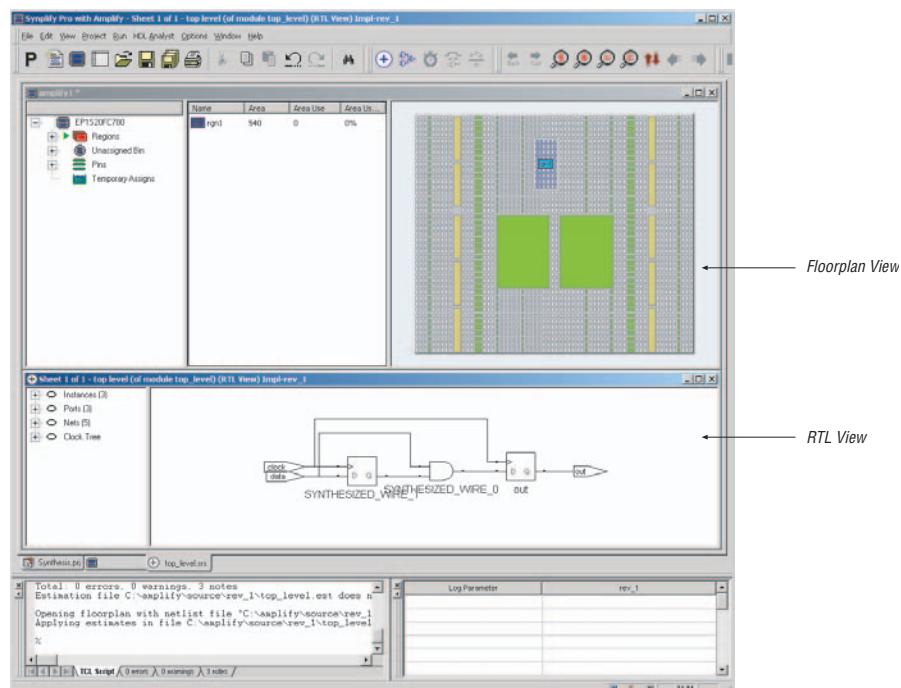
You must determine which paths to constrain based on the critical paths from the previous implementation. When Quartus II projects are launched with the Amplify Tcl script, the Quartus II software generates a `<project name>.tan.rpt` file that lists the critical paths for the design. You

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 will appear (see [Figure 11–3](#)).

Figure 11–3. 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:

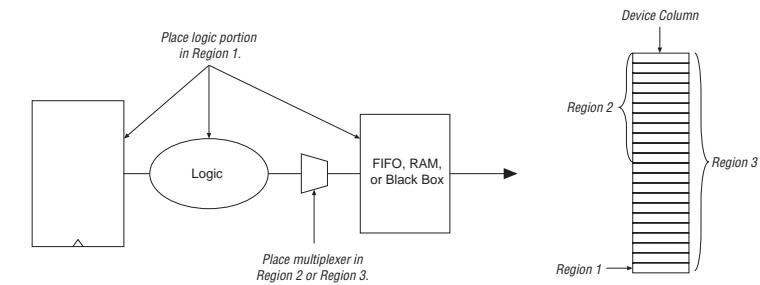
1. Create a region in the Amplify Physical Optimizer device floorplan window and select the module in the RTL view of the design.
2. Drag the module to the new region. The software will then report the utilization of the region.
3. Resynthesize the design in the software to reoptimize the critical path 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.

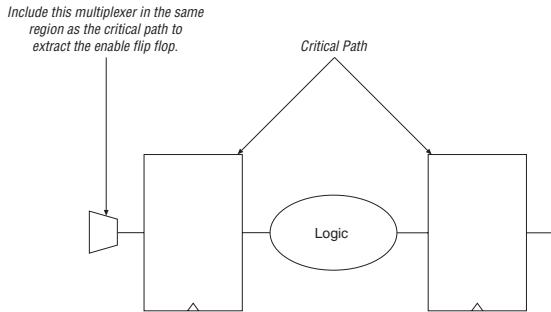
Multiplexers

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 11–4](#) shows placing critical paths with multiplexers.

Figure 11–4. Placing Critical Paths with Multiplexers

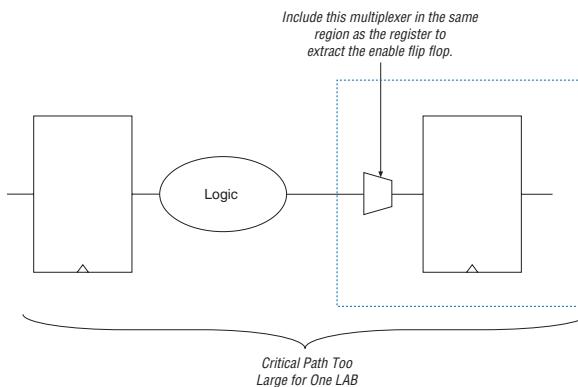


If the critical path contains a multiplexer feeding a register, create a region and place the multiplexer along with the entire critical path in the region. See [Figure 11–5](#).

Figure 11–5. 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 11–6 shows large critical paths with multiplexers feeding registers.](#)

Figure 11–6. 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.

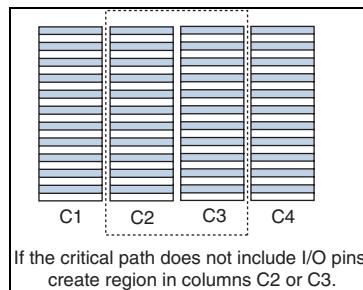
3. Assign the second critical path to the second region.

Feedback Paths

If critical paths have the same start and end points, follow the steps below in the Amplify Physical Optimizer software (see [Figure 11–7](#)):

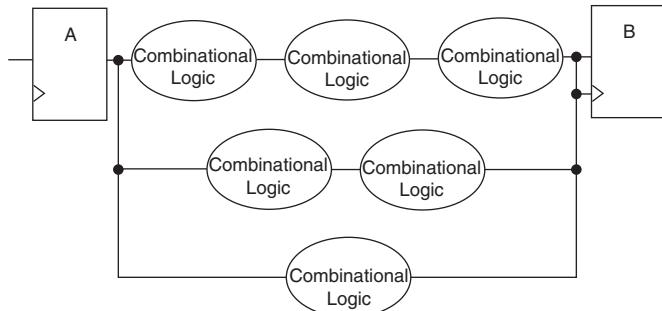
1. Select the register and instance not directly connected to the register.
2. Select **Filter Schematic** twice (right-click menu).
3. Highlight the line leading out of the register and either press **P** or right-click the line. Select **Expand Paths**. Assign this logic to a region.

Figure 11–7. Critical Paths with the Same Starting or Ending Points

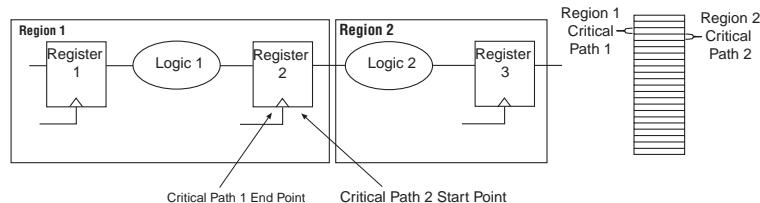


Starting and Ending Points

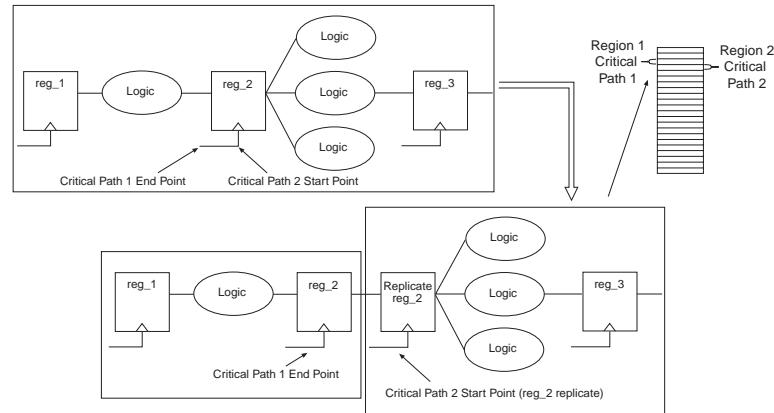
[Figure 11–8](#) shows a critical path that has multiple starting and ending points. Use **Find** to display all the starting and ending points in the RTL view in Amplify. Expand the paths between those points. If there is unrelated logic between the multiple starting points and ending points, assign 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 11–8. 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 11–9](#) shows two critical paths that share a register.

Figure 11–9. Two Critical Paths Sharing a Register

If the fanout is on the shared region, replicate the register and assign both registers to two regions (see [Figure 11–10](#)). This is done by dragging the same register to the required regions. Entities and nodes are also replicated by performing the same procedure.

Figure 11–10. 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. You can also update the utilization estimates by using the estimate region feature by selecting **Estimate Area** (Run menu).

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 11–1 provides a description of each script's purpose.

Table 11–1. Amplify Physical Optimizer Tcl Script Description	
Tcl File	Description
<i><project name>_cons</i>	This Tcl script will create and compile a Quartus II project. The <i><project name>.tcl</i> will automatically be sourced when this script is sourced.
<i><project name></i>	This script contains forward annotation of constraint information including clock frequency, duty cycle, location, etc.
<i><project name>_rm</i>	This script removes any previous constraints from the project. The removed constraint is saved in <i><project name>_prev.tcl</i>

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.

```
<commnd prompt>quartus_cmd my_project_cons.tcl ↵
```

 You must execute 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.



See 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 synthesizable 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 synthesizable 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.

This clear box can be turned on by checking the **Generate Clearbox body (for EDA tools only)** option in the **MegaWizard Plug-In Manager** (Tools menu) 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 11-14](#).

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** (Tools menu) 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 see 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.

A

Amplify

- Physical Synthesis Concepts 11–1
- to-Quartus II Flow 11–2
- Using Amplify Physical Optimizer
 - Floorplans 11–5
- Using the Amplify Physical Optimizer
 - Floorplans 11–5

Archive

- a Project 4–7
- All Compilations 9–15
- Projects With a Tcl Command or at the Command Prompt 4–14
- Projects With the Quartus II Archive Project Feature 4–6
- Area Optimization 9–10
- Assignment Editor 5–9
 - Category 1–2
 - Customizable Columns 1–12
 - Features 1–3, 1–8
 - Pin Assignment Location 5–9
 - Settings Made Outside User Interface 1–1
 - Using 1–1
 - Using to Place Logic 10–15
- Assignments 1–8
 - Back-Annotating 8–17
 - Dynamic Syntax Checking 1–9
 - Exporting 1–8, 1–16
 - Exporting & Importing 1–16
 - Import 1–16, 1–18, 10–21
 - Importing 1–18
 - Location 5–13
 - Path-Based 10–25
 - Revisions 4–1

Atom Netlist

- Design Information 10–19
- Specify 10–20

B

Back-Annotation 10–41

- Assignments 8–17
- LogicLock Regions 10–41
- Manual Placement 6–41
- Routing Information 10–34
- Backlash Substitution 3–3
- Bar
 - Category 1–2
 - Edit 1–3
 - Information 1–6
 - Node Filter 1–5
- Black Box
 - Methodology, Floorplans 11–14
- Block-Based Design
 - Quartus II LogicLock Methodology 10–2
- Category
 - Bar 1–3
 - Node Filter, Information, Edit Bars & Spreadsheet 1–3
- Clear Box
 - Methodology, Floorplans 11–13
- Clock
 - Speed
 - Improving in Design 6–58
 - to-Output Time
 - Improving 6–25
 - Using Fast Regional in Stratix Devices 6–30
- Columns
 - Customizable 1–7
- Combinational Logic
 - Physical Synthesis 8–10
- Command Line
 - Accessing Arguments 3–13
 - Arguments 3–13
 - cmdline Package 3–14
 - Evaluate as Tcl 3–12
- Option
 - t, -s, and --tcl_eval 3–11
- Option, Details 2–9

Options 2–9 Scripting Examples 2–11 Scripting Help 2–7 Compilation Failure Continue Exploration 9–15 Initial Settings 6–5 Restore Original Results 4–6 Time 6–13 Time Optimization Techniques 6–60 Timing Driven 6–5 with quartus_sh --flow 2–7 Compile Archive 9–15 Verify the Top-Level Design 10–23 Compile Designs 3–17 Compilation Time Optimization Techniques 6–60 Constraint Create a Project and Apply 2–11 Priority 10–31 Remove Fitter 6–19 Creating Custom Spaces for DSE 9–17 Different Versions of Your Design 4–5 LogicLock Regions 10–4 Modifying LogicLock Regions 10–38 Pin Locations Using the Assignment Editor 1–13 Projects & Making Assignments 3–16 Creating, Revisions 4–2 Critical Path Reducing Delay 6–43 Custom Space Simple 9–20 XML Schema 9–21	Design Analysis 6–8 Compile 3–17 Creating Different Versions 4–5 File Check Syntax 2–11 Fit Quickly 2–14 Fit Using Multiple Seeds 2–14 Flow 5–1 Hierarchy 10–9 Improving Performance 6–34 Optimization Improve Resource Utilization 6–14 Optimization for Altera Devices 6–1 Optimize 6–5 Optimizing Compilation Time 6–60 Using Revisions 4–1 Verify 10–23 Design Analysis 6–8 Using the Timing Closure Floorplans 7–1 Design Flow Bar, Node Filter 1–10 Complete Design Files 5–4 Design Files 5–4 No Design Files 5–2 Partial Design Files 5–4 Without Design Files 5–2 Design Space Explorer 6–2 DESIGNSPACE 9–18 Tag 9–18 Device Using Larger 6–24 Device Resources Reserve 6–47 Drag & Drop Using to Place Logic 10–14 DSE Advanced Information 9–16 Advanced Search Options 9–7 Archive Compiles 9–15 Command Line Options 9–2 Computer Load Sharing 9–16 Concepts 9–1 Creating Custom Spaces 9–17 Distributed Using LSF 9–16 Distributed Using Quartus II Master
---	--

D

Databases
Exporting 4–9
Importing 4–15
Version-Compatible 4–15
Delays
Programmable 6–27

-
- Process 9–16
 - Exploration 9–1
 - Settings 9–4
 - Space 9–8 - Flow 9–4
 - Options 9–13
 - General Information 9–2
 - LogicLock Region Restructuring 9–7
 - Optimization 9–7
 - Performance Options 9–7
 - Performing Advanced Search 9–8
 - Project Settings 9–6
 - Seed & Seed Sweeping 9–1
 - Support for Altera Device Families 9–5
 - Support for Altera device Families 9–5
 - DSE Advanced Information 9–16
 - DSP Blocks
 - Retarget 6–22
 - Dynamic Syntax Checking 1–9
-
- ## E
- Early Timing Estimation 6–5
 - EDA
 - Tool Assignments 3–1
 - Edit Bars 1–2
 - Entity
 - Analysis & Synthesis Resource
 - Utilization 10–25
 - Executables
 - Supporting Tcl 3–11
 - Exploration
 - Base Compile Failure 9–15
 - Run Quartus Assembler 9–15
 - Space 9–8
 - Area Optimization Search 9–10
 - Custom Space 9–10
 - Exploration Point 9–1
 - Extra Effort Search 9–9
 - Retiming Search 9–10
 - Save to File 9–15
 - Stop Flow After Time 9–15
 - Extracting, Report Data 3–18
-
- ## F
- Fan-In
-
- Estimating 6–49
 - Fan-In When Assigning Output Pins 6–49
 - Fan-Out Control 6–36
 - Duplicate Logic 6–36
 - Fast Input, Output & Output Enable Registers 6–26
 - Fast Regional Clocks
 - Using in Stratix Devices 6–30
 - Fit a Design as Quickly as Possible 2–14
 - Fitter
 - Effort Setting 6–7
 - Floorplans 11–11
 - Amplify
 - Forward Annotating Physical Optimizer Constraints into the Quartus II Software 11–12
 - Physical Optimizer Constraints
 - Forward Annotating 11–12
 - Software 11–13
 - Black Box
 - Methodology 11–14
 - Clear Box
 - Methodology 11–13
 - Detailed 11–11
 - Path
 - Feedback 11–9
 - Independent 11–8
 - Timing Closure 7–1
 - Using MegaWizard Plug-In Manager with Amplify 11–13
 - fMAX
 - Improving 6–31
 - Timing 6–11
 - Analysis Report 7–18
 - Optimization Techniques 6–31
-
- ## G
- Gate-Level Register Retiming 8–4
 - Generating a Mapped Netlist 5–10
 - Global Control Signals
 - Dedicated Inputs 6–46
-
- ## H
- Hierarchy

- Assignments 5–1, 6–38
 Flatten 6–35
 Window 10–9
 Hold Times 6–30
- I**
- I/O
 Assignment
 Analysis 5–2, 5–12
 Anaylsis 5–1
 Anaysis
 Tcl Command 5–12
 Creating 5–1
 Design Flow 5–1
 Inputs Used for Analysis 5–8
 Placement 5–10
 Planning 5–1
 Understanding Analysis Report 5–11
 Pin Reserving 5–8
 Timing 6–7, 6–10
 Optimization Techniques 6–24, 6–66
 Using a PLL to Improve 6–29
- Importing
 Back-Annotated Routing in LogicLock
 Regions 10–36
 Exporting
 Version-Compatible Databases 4–15
 LogicLock Regions 10–40
 the Assignments 10–21
- Improving
 Design Performance 10–1
 fMAX Summary 6–31
 Maximum Frequency (fMAX) 6–58
 Setup & Clock-to-Output Times
 Summary 6–25
- Incremental Synthesis 6–61
 Information 1–2
 Initial Pass
 No Physical Constraints 11–3
 Initializing and Uninitializing a LogicLock
 Region 10–38
 Inputs for I/O Assignment Analysis 5–8
 Interactive Shell Mode 2–14
 Iterative Passes
 Optimizing the Critical Paths 11–5
- L**
- LCELL Buffers
 Using to Reduce Required Resources 6–53
 Load Sharing in DSE Using Distributed Exploration Searches 9–16
 Location Assignments
 & Back-Annotation 6–40
 Logic Lock
 Region
 Properties 10–11
 LogicLock
 Additional Quartus II Design Features 10–23
 Assignment Precedence 10–27
 Assignments 6–37
 Location & Back Annotation 6–40
 Back Annotation 6–41
 Connection Count 10–25
 Constraint Priority 10–31
 Critical Paths 10–25
 Design Features 10–4
 Design Flow 10–41
 Design Hierarchy 10–9
 Design Methodology 10–1
 Designing with 10–4
 Drag & Drop 10–14
 DSE 9–7
 Exporting 10–40
 Exporting Back-Annotated Routing 10–34
 Hierarchical (Parent and/or Child) 10–12
 Importing
 the Module 10–20
 Improving Design Performance 10–1
 Manual Placement 6–41
 Module
 Export 10–18
 Import 10–20
 Optimize 10–17
 Synthesize 10–17
 Path Assignments 6–39
 Region
 Assigning Content 10–13
 Back-Annotating
 Exporting 10–34
 Importing 10–36
 Back-Annotating Routing
 Information 10–34

-
- Connectivity 10–24
 - Exporting 10–40
 - Hierarchical 10–12
 - Importing 10–40
 - Initializing 10–38
 - Modifying 10–38
 - Obtaining Properties 10–38
 - Placing 10–31
 - Placing Memory 10–32
 - Placing Other Device Features 10–32
 - Placing Pins 10–32
 - Prevent Assignment Option 10–24
 - Properties 10–4
 - Reserve 10–23
 - Specify Size and Location 10–11
 - Tcl Command 10–10
 - Tcl Scripts 10–17
 - Uninitializing 10–38
 - Viewing Routing Congestion 7–14
 - Window 10–4
 - Region Content 10–39
 - Region Creating 10–38
 - Regions 6–62
 - Back Annotating 6–40
 - Back-Annotated 6–64
 - Custom 6–41
 - Hierarchical 6–38
 - Soft LogicLock Regions 10–28
 - Regions 5–2
 - Resource Utilization 10–25
 - Restrictions 10–31
 - Revisions Feature 10–27
 - Routing 10–20
 - Setting Assignment Priority 10–40
 - Tooltips 10–23
- ## M
- Macrocell Usage
 - Resolving Issues 6–51
 - Makefile
 - Implementation 2–16
 - Managing Revisions 4–14
 - Mapped Netlist
 - Generating 5–10
 - Maximum Frequency 6–58
 - Memory Blocks
 - Retarget 6–21
- Modify Pin Assignments or Choose a Larger Package 6–24
 - Modular Executables 2–7
 - Makefile 2–16
 - Report Files 2–6
 - Module Performance
 - Preserving 10–4
- ## N
- Netlist
 - Optimization 8–1
 - Applying Options 8–15
 - Netlist Optimization Options 8–15
 - Node Filter 1–2
 - Node Filter Bar 1–10
 - Node-level Netlist
 - Save into Persistent Source File 10–39
- ## O
- Optimization
 - Advisors 6–2
 - Goal 9–12
 - Techniques
 - Resource Utilization 6–14
 - Optimize Source Code 6–23
 - Optimizing
 - Critical Path 6–43
 - Placement
 - Cyclone Devices 6–44
 - Mercury, APEX II, APEX 20KE/C Devices 6–45
 - Stratix Family Devices & Cyclone II Devices 6–43
 - Output Pins
 - Estimate Fan-In When Assigning 6–49
- ## P
- Parallel Expanders
 - Used Within a LAB 6–50
 - Path
 - Assignments 6–39
 - Physical Synthesis
 - Combinational Logic 8–10
 - Optimization 8–17
 - Optimization 6–32

- Preserving Results 8–13
 - Register Retiming 8–13
 - Report 8–10
 - Pin
 - Assignment
 - Modify 6–24
 - Pin Assignment
 - Output Enable 6–48
 - Pin Assignment
 - Control Signal 6–48
 - Guidelines & Procedures 6–47
 - Minimize Fitting Issues 6–47
 - Outputs Using Parallel Expander 6–49
 - Pin Assignments
 - Control Signal 6–48
 - Pipling
 - Complex Register Logic 6–59
 - PLL
 - Using to Shift Clock Edges 6–29
 - Projects
 - Archiving 4–6
 - Creating 3–16
 - Making Assignments 3–16
 - Restoring Archived 3–16, 4–15
 - Propagation Delay 6–57
- Q**
- QFlow Script 2–18
 - QSF
 - Specify 10–20
 - Quartus II
 - Megafunctions
 - Using MegaWizard Plug-In Manager with Amplify 11–13
 - MegaWizard Plug-In
 - Manager-generated Verilog HDL Files for Clear Box Megafunction
 - Instantiation 11–13
 - MegaWizard Plug-In
 - Manager-generated VHDL Files for Clear Box Megafunction
 - Instantiation 11–13
 - Modular Executables 2–1, 2–2, 2–20
 - Tcl API Help 3–8
 - Tcl Packages 3–9

Quartus II

 - Block-Based Design Flow 10–17
 - Legacy Tcl Support 3–33
 - LogicLock
 - Design Features 10–23
 - Revisions Feature 10–27
 - quartus_sh --flow
 - Compilation 2–7

R

 - Register
 - Retiming to Trade-Off tSU/tCO With fMAX 8–8
 - Register Packing 6–16
 - Register Retiming
 - Gate-Level 8–4
 - Physical Synthesis for Registers 8–13
 - Trade-Off tSU/tCO with fMAX 8–8
 - Registers
 - Fast Input, Output & Output Enable 6–26
 - Report Data
 - Extracting 3–18
 - Resource Utilization 6–9
 - Analysis & Synthesis by Entity 10–25
 - Optimization Techniques 6–14, 6–46, 6–65
 - Resolving Issues 6–14
 - Resolving Problems 6–50
 - Restoring Archived Projects 4–15
 - Resynthesis
 - Perform WYSIWYG for Area 6–19
 - WYSIWYG Primitive 8–2
 - Revision
 - Comparing 4–5
 - Creating 4–1
 - Deleting 4–2
 - Revisions
 - Creating & Deleting 4–2
 - Create From a DSE Point 9–13
 - Routing
 - Congestion 6–64, 7–14
 - Resolving Issues 6–52

S

 - Seed
 - Sweep 9–9
 - Extra Effort Search 9–9

Sweeping 9–1
Seeds
Fit a Design 2–14
Settings
Fitter Effort 6–7, 6–62
Initial Compilation 6–65
Physical Synthesis Effort 6–62
Smart Compilation 6–5
Initial Compilation 6–65
Setup Time
Improving 6–55
Source Code
Optimize 6–23, 6–45
Optimizing Using Pipelining Technique 6–59
Spreadsheet 1–2
State Machine
Encoding 6–36
State Machine Encoding
Change 6–36
Synthesis
Flatten the Hierarchy 6–35
Netlist Optimization 6–32, 8–2, 8–4
Optimize for Area 6–20
Optimize for Speed 6–34
Options, Other 6–37
Reduce Netlist Optimization Time 6–60
Reducing 6–60
Set Effort to High 6–35
Specify State Machine Encoding 6–20
Synthesize
Netlist Using Netlist Optimizations 2–14

T

Tcl
API Reference 3–8
Assignment 6–65
Back-Annotate Command 8–17
Command 5–13
Commands Collection 3–19
Console Window 3–12
Control Structures 3–5
Defined 3–1
Evaluate 3–12
Executables 3–11
Getting Help 2–20

Getting Help on Tcl & Tcl APIs 3–31
Help 3–2
List 3–4
Loading Packages 3–10
Packages 2–9
Procedure 3–6
Quartus II Legacy Support 3–33
Quartus II Legacy Support 3–2
Script
Run 4–14
Scripting Basics 3–2
Scripts 10–10
Shell in Interactive Mode 3–28
Tk
GUI Help Interface 3–2
Timing
fMAX Optimization Techniques 6–31
Optimization Techniques 6–54
Results
Preserving Using LogicLock Flow 10–3
Timing Analysis 3–20
Timing Closure
Design Anlaysis 7–21
Floorplans 7–1
Assigning LogicLock Region Content 10–13
Design Anlaysis 7–1
Editor, Physical Timing Estimates 7–10
Viewing Assignments 7–3
Viewing Critical Paths 7–6
Floorplans Editor 10–8
LogicLock Regions Connectivity 10–24
View 10–10
Floorplans Views 7–2
Timing Estimation Before Fitting 6–61
Timing Optimization 6–57
Clock-to-Output Time 6–56
Improving Propagation Delay 6–57
Maximum Frequency 6–58
Propagation Delay (tPD) 6–57
Setup Time 6–55

V

VHDL Files
Clear Box Megafunction Instantiation 11–13
Virtual Pins 10–41

Assigning [10–41](#)



Quartus II Handbook, Volume 3

Verification

ALTERA[®]

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
<http://www.altera.com>

qii5v3_3.1

Copyright © 2005 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.



Printed on recycled paper





Contents

Chapter Revision Dates	xi
-------------------------------------	-----------

About this Handbook	xiii
----------------------------------	-------------

How to Contact Altera	xiii
-----------------------------	------

Typographic Conventions	xiii
-------------------------------	------

Section I. Simulation

Revision History	Section I-2
------------------------	-------------

Chapter 1. Mentor Graphics ModelSim Support

Introduction	1-1
Background	1-1
Software Compatibility	1-2
Altera Design Flow with ModelSim-Altera Software	1-3
Functional RTL Simulation	1-4
Functional RTL Simulation Libraries	1-4
Simulating VHDL Designs	1-5
Simulating Verilog Designs	1-8
Post-Synthesis Simulation	1-12
Simulating VHDL Designs	1-13
Simulating Verilog Designs	1-16
Gate-Level Timing Simulation	1-18
Quartus II Software Output Files for use in the ModelSim-Altera Software	1-19
Gate Level Simulation Libraries	1-19
Simulating VHDL Designs	1-21
Simulating Verilog Designs	1-24
Using the NativeLink Feature with ModelSim	1-27
Scripting Support	1-28
Generating a Post-Synthesis Simulation Netlist for Modelsim	1-28
Generating a Gate-Level Timing Simulation Netlist for VCS	1-29
Software Licensing & Licensing Set-Up	1-29
LM_LICENSE_FILE Variable	1-30
Conclusion	1-30

Chapter 2. Synopsys VCS Support

Introduction	2-1
Software Requirements	2-1
Using VCS in the Quartus II Design Flow	2-1
Functional RTL Simulations	2-3

Post-Synthesis Simulation	2-5
Gate-Level Timing Simulation	2-7
Common VCS Compiler Options	2-9
Using VirSim: The VCS Graphical Interface	2-10
VCS Debugging Support VCS Command-Line Interface	2-10
Using PLI Routines with the VCS Software	2-10
Preparing & Linking C Programs to Verilog Code	2-10
Scripting Support	2-11
Generating a Post-Synthesis Simulation Netlist for VCS	2-12
Generating a Gate-Level Timing Simulation Netlist for VCS	2-12
Conclusion	2-13

Chapter 3. Cadence NC-Sim Support

Introduction	3-1
Software Requirements	3-1
Operation Modes	3-3
Quartus II Software & NC Simulation Flow Overview	3-4
Functional/RTL Simulation	3-5
Set Up Your Environment	3-5
Create Libraries	3-6
Simulating a Design With Memory	3-11
Compile Source Code & Testbenches	3-12
Elaborate Your Design	3-14
Add Signals to View	3-16
Simulate Your Design	3-19
Post-Synthesis Simulation	3-20
Quartus II Simulation Output Files	3-20
Set Up Your Environment	3-21
Create Libraries	3-21
Compile the Project Files & Libraries	3-22
Elaborate Your Design	3-22
Add Signals to View	3-22
Simulate Your Design	3-22
Gate-Level Timing Simulation	3-22
Quartus II Simulation Output Files	3-22
Quartus II Timing Simulation Libraries	3-24
Set Up Your Environment	3-24
Create Libraries	3-24
Compile the Project Files & Libraries	3-25
Elaborate Your Design	3-25
Add Signals to View	3-27
Simulate Your Design	3-27
Incorporating PLI Routines	3-27
Dynamically Link a PLI Library	3-28

Dynamically Load a PLI Library	3–29
Statically Link the PLI Library With NC-Sim	3–32
Scripting Support	3–33
Generate NC-Sim Simulation Output Files	3–34
References	3–35

Section II. Timing Analysis

Revision History	Section II–1
------------------------	--------------

Chapter 4. Simulating Altera IP in Third-Party Simulation Tools

Introduction	4–1
Generating an IP Functional Simulation Model with IP Toolbench	4–1
Low-Level VO or VHO Simulation Models	4–2
High-Level VO or VHO Simulation Models	4–3
Launch IP Toolbench	4–4
Step 1: Parameterize	4–5
Step 2: Set Up Simulation	4–5
Step 3: Generate	4–6
Step 4: Instantiate the IP Functional Simulation Model in Your Design	4–8
Step 5: Perform Simulation	4–8
Design Language Examples	4–10
Verilog Example: Simulating the IP Functional Simulation Model in the ModelSim Software ..	
4–10	
VHDL Example: Simulating the IP Functional Simulation Model in the ModelSim Software ...	
4–11	
NC-VHDL Example: Simulating the IP Functional Simulation Model in the NC-VHDL	
Software	4–13
Verilog HDL Example: Simulating Your IP Functional Simulation Model in VCS	4–14
Conclusion	4–15

Chapter 5. Quartus II Timing Analysis

Introduction	5–1
Timing Analysis Basics	5–1
Clock Setup Time (tSU)	5–1
Clock Hold Time (tH)	5–2
Clock-to-Output Delay (tCO)	5–3
Pin-to-Pin Delay (tPD)	5–3
Maximum Clock Frequency (fMAX)	5–3
Slack	5–4
Hold Time Slack	5–4
Clock Skew	5–5
Setting up the Timing Analyzer	5–6
Setting Global Timing Assignments	5–6
Specifying Individual Clock Requirements	5–7
Setting Other Individual Timing Assignments	5–8

Advanced Timing Analysis	5–14
Clock Skew	5–14
Multiple Clock Domains	5–16
Multicycle Paths	5–17
Typical Applications of Multicycle Assignments	5–20
False Paths	5–29
Fixing Hold Time Violations	5–32
Timing Analysis Across Asynchronous Domains	5–32
Best Case Timing Analysis	5–33
Best Case Timing Analysis Settings	5–33
Performing Best Case Timing Analysis	5–34
Best Case Timing Analysis Reporting	5–34
Recovery and Removal Analysis	5–34
Recovery Report	5–34
Removal Report	5–35
Early Timing Estimation	5–36
Latch Analysis	5–37
General Timing Analyzer Commands	5–38
Creating a Timing Netlist	5–38
Advanced Timing Analysis & Reports	5–39
Conclusion	5–41

Section III. Power Estimation & Analysis

Revision History	Section III–1
------------------------	---------------

Chapter 6. Synopsys PrimeTime Support

Introduction	6–1
Quartus II Settings for Generating PrimeTime Files	6–2
The Netlist	6–3
The Standard Delay Output File	6–3
The Tcl Script	6–4
Sample of Constraints Specified in PrimeTime Format	6–4
Running the PrimeTime Software	6–5
Analyzing Quartus II Projects	6–5
Other pt_shell Commands	6–5
PrimeTime Timing Reports	6–6
Sample PrimeTime Timing Report	6–6
Conclusion	6–7

Chapter 7. PowerPlay Early Power Estimator

Introduction	7–1
PowerPlay Early Power Estimator Spreadsheet	7–1
Estimating Power in the Design Cycle	7–4
Quartus II Early Power Estimator File	7–6
Conclusion	7–8

Section IV. On-Chip Debugging

Revision History	Section IV-1
------------------------	--------------

Chapter 8. PowerPlay Power Analyzer

Types of Power Analyses	8-3
Factors Affecting Power Consumption	8-3
Device Selection	8-3
Environmental Conditions	8-4
Design Resources	8-5
Signal Activities	8-5
PowerPlay Power Analyzer Flow	8-6
Operating Conditions	8-7
Signal Activities Data Sources	8-8
Using the PowerPlay Power Analyzer	8-13
Common Analysis Flows	8-13
Generating a Signal Activity File Using the Quartus II Simulator	8-14
Generating a Value Change Dump File Using a Third-Party Simulator	8-17
Running the PowerPlay Power Analyzer Using the Quartus II GUI	8-19
Scripting Support	8-28
Conclusion	8-29

Chapter 9. Quick Design Debugging Using SignalProbe

Introduction	9-1
Using SignalProbe	9-1
Reserving SignalProbe pins	9-2
Adding SignalProbe Sources	9-3
Assigning I/O Standards	9-5
Adding Registers for Pipelining	9-5
Performing a SignalProbe Compilation	9-7
Running SignalProbe with Smart Compilation	9-8
Analyzing SignalProbe Routing Failures	9-8
Understanding the Results of a SignalProbe Compilation	9-10
Scripting Support	9-11
Reserving SignalProbe Pins	9-11
Adding SignalProbe Sources	9-12
Assigning I/O Standards	9-12
Adding Registers for Pipelining	9-12
Run SignalProbe Automatically	9-13
Run SignalProbe Manually	9-13
Enable or Disable All SignalProbe Routing	9-13
Running SignalProbe with Smart Compilation	9-13
Allow SignalProbe to Modify Fitting Results	9-14
Conclusion	9-14

Chapter 10. Design Debugging Using the SignalTap II Embedded Logic Analyzer

Introduction	10-1
--------------------	------

Including the SignalTap II Logic Analyzer in Your Design	10-3
Using the STP File to Create an Embedded Logic Analyzer	10-4
Using the MegaWizard Plug-In Manager to Create Your Embedded Logic Analyzer	10-10
Programming the Device for SignalTap II Analysis	10-13
View Data Samples	10-14
Advanced Features	10-15
Preserving FPGA Memory	10-15
Creating Complex Triggers	10-16
Using External Triggers	10-19
Embedding Multiple Analyzers in One FPGA	10-22
Faster Compilations	10-23
Time Bars & Next Transition	10-24
Saving Captured Data	10-25
Converting Captured Data to Other File Formats	10-25
Creating Mnemonics for Bit Patterns	10-25
Segmenting Your Sample Depth	10-26
Capturing Data to a Specific RAM Type	10-27
FPGA Resources Used by SignalTap II	10-27
Using SignalTap II in a Lab Environment	10-28
Managing Multiple STP Files	10-28
Remote Debugging Using the SignalTap II Logic Analyzer	10-29
Signal Preservation	10-32
Tappable Signals	10-33
Timing Preservation with SignalTap II Logic Analyzer	10-33
Using SignalTap II Logic Analyzer to Simultaneously Debug Multiple Designs	10-34
Locating a Node in the Chip Editor	10-34
SignalTap II Scripting Support	10-35
SignalTap II Command Line Options	10-35
SignalTap II Tcl Commands	10-36
Design Example: Preserving Timing	10-38
Design Example: Using SignalTap II Logic Analyzers in SOPC Builder Systems	10-41
Conclusion	10-41

Chapter 11. Design Analysis & Engineering Change Management with Chip Editor

Introduction	11-1
Background	11-1
Using the Chip Editor in Your Design Flow	11-2
Chip Editor Features	11-4
Chip Editor Floorplan	11-4
First (Highest) Level View	11-5
Second Level View	11-7
Third Level View	11-8
Bird's Eye View	11-9
Resource Property Editor	11-11

LE Properties	11–11
ALM Properties	11–15
FPGA I/O Elements	11–18
Modifying the PLL Using the Chip Editor	11–23
Change Manager	11–25
Complex Changes in the Change Manager	11–27
Common Applications	11–28
Routing an Internal Signal to an Output Pin	11–28
Adjust the Phase Shift of a PLL to Meet I/O Timing	11–29
Correcting a Design Flaw	11–29
Example Design: Meeting I/O Timing	11–29
Running the Quartus II Timing Analyzer	11–36
Generating a Netlist for Other EDA Tools	11–36
Generating a Programming File	11–36
Conclusion	11–36

Section V. Formal Verification

Revision History	Section V–2
------------------------	-------------

Chapter 12. In-System Updating of Memory & Constants

Introduction	12–1
Overview	12–1
Device & Megafunction Support	12–2
Creating In-System Modifiable Memories & Constants	12–3
Running the In-System Memory Content Editor	12–4
Instance Manager	12–5
Editing Data Displayed in the Hex Editor	12–7
Importing & Exporting Memory Files	12–7
Viewing Memories & Constants in the Hex Editor	12–7
Programming the Device Using the In-System Memory Content Editor	12–9
Example: Using the In-System Memory Content Editor with the SignalTap II Embedded Logic Analyzer	12–9
Conclusion	12–10

Chapter 13. Cadence Incisive Conformal Equivalency Checker Support

Introduction	13–1
Formal Verification	13–1
Equivalence Checking	13–1
Formal Verification Support	13–2
Quartus Integrated Synthesis	13–2
Synplify Pro	13–3
RTL Coding for Quartus Integrated Synthesis (QIS)	13–3
Synthesis Directives & Attributes	13–4
Stuck-at Registers	13–5
ROM & Shift Register Inference	13–6

Latch Inference	13–6
Combinational Loops	13–6
Finite State Machine Coding Styles	13–6
Generating the VO File & Incisive Conformal Script	13–7
Quartus II Scripts for LEC	13–14
Comparing Designs Using Incisive Conformal Software	13–16
Black Boxes in the Incisive Conformal Flow	13–16
Running the Incisive Conformal Software	13–17
Known Issues & Limitations	13–19
Conclusion	13–19

Chapter 14. Synopsys Formality Support

Introduction	14–1
Formal Verification	14–1
Equivalence Checking	14–1
Formal Verification Support	14–2
EDA Tools & Device Support	14–2
Formal Verification Between RTL & Post-Synthesis Netlist	14–2
Generating Post-Synthesis Netlist for Formal Verification	14–3
DC FPGA Software Settings	14–3
Generating the VO File & Formality Script	14–4
Handling Black Boxes	14–8
Quartus II Scripts for Formality	14–11
Comparing Designs Using Formality Software	14–11
Known Issues & Limitations	14–12
Conclusion	14–12
Related Links	14–12
Tcl Sample Script	14–13
DC FPGA Synthesis Script	14–13
Quartus II Software Generated Formal Verification Script	14–14

Index



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. Mentor Graphics ModelSim Support
Revised: December 2004
Part number: *qii53001-3.0*
- Chapter 2. Synopsys VCS Support
Revised: December 2004
Part number: *qii53002-2.1*
- Chapter 3. Cadence NC-Sim Support
Revised: December 2004
Part number: *qii53003-3.0*
- Chapter 4. Simulating Altera IP in Third-Party Simulation Tools
Revised: December 2004
Part number: *qii53014-1.0*
- Chapter 5. Quartus II Timing Analysis
Revised: January 2005
Part number: *qii53004-2.2*
- Chapter 6. Synopsys PrimeTime Support
Revised: December 2004
Part number: *qii53005-2.1*
- Chapter 7. PowerPlay Early Power Estimator
Revised: December 2004
Part number: *qii53006-2.1*
- Chapter 8. PowerPlay Power Analyzer
Revised: December 2004
Part number: *qii53013-1.0*
- Chapter 9. Quick Design Debugging Using SignalProbe
Revised: December 2004
Part number: *qii53008-2.1*

Chapter 10. Design Debugging Using the SignalTap II Embedded Logic Analyzer

Revised: December 2004

Part number: *qii53009-2.1*

Chapter 11. Design Analysis & Engineering Change Management with Chip Editor

Revised: January 2005

Part number: *qii53010-2.1*

Chapter 12. In-System Updating of Memory & Constants

Revised: December 2004

Part number: *qii53012-1.2*

Chapter 13. Cadence Incisive Conformal Equivalency Checker Support

Revised: December 2004

Part number: *qii53011-2.1*

Chapter 14. Synopsys Formality Support

Revised: January 2005

Part number: *qii53015-1.0*



About this Handbook

This handbook provides comprehensive information about the Altera® Quartus® II design software, version 4.2.

How to Contact Altera

For the most up-to-date information about Altera products, go to the Altera world-wide web site at www.altera.com. For technical support on this product, go to www.altera.com/mysupport. For additional information about Altera products, consult the sources shown below.

Information Type	USA & Canada	All Other Locations
Technical support	www.altera.com/mysupport/	altera.com/mysupport/
	(800) 800-EPLD (3753) (7:00 a.m. to 5:00 p.m. Pacific Time)	(408) 544-7000 (1) (7:00 a.m. to 5:00 p.m. Pacific Time)
Product literature	www.altera.com	www.altera.com
Altera literature services	(1)	(1)
Non-technical customer service	(800) 767-3753	(408) 544-7000 (7:30 a.m. to 5:30 p.m. Pacific Time)
FTP site	ftp.altera.com	ftp.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 below.

Visual Cue	Meaning (Part 1 of 2)
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: f_{MAX} , \qdesigns directory, d: drive, chiptrip.gdf file.
<i>Italic Type with Initial Capital Letters</i>	Document titles are shown in italic type with initial capital letters. Example: <i>AN 75: High-Speed Board Design</i> .

Visual Cue	Meaning (Part 2 of 2)
<i>Italic type</i>	<p>Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$.</p> <p>Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: <file name>, <project name>.pof file.</p>
Initial Capital Letters	<p>Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.</p>
"Subheading Title"	<p>References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: "Typographic Conventions."</p>
Courier type	<p>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.</p> <p>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.</p>
1., 2., 3., and a., b., c., etc.	<p>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.</p>
	<p>Bullets are used in a list of items when the sequence of the items is not important.</p>
	<p>The checkmark indicates a procedure that consists of one step only.</p>
	<p>The hand points to information that requires special attention.</p>
 <small>CAUTION</small>	<p>The caution indicates required information that needs special consideration and understanding and should be read prior to starting or continuing with the procedure or process.</p>
	<p>The warning indicates information that should be read prior to starting or continuing the procedure or processes</p>
	<p>The angled arrow indicates you should press the Enter key.</p>
	<p>The feet direct you to more information on a particular topic.</p>



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, Mentor Graphics ModelSim Support](#)
- [Chapter 2, Synopsys VCS Support](#)
- [Chapter 3, Cadence NC-Sim Support](#)

Revision History

The table below shows the revision history for [Chapter 1](#) to [3](#).

Chapter(s)	Date / Version	Changes Made
1	Dec. 2004 v3.0	<ul style="list-style-type: none">● Reorganized chapter and updated information.● Updated tables and figures.● New functionality for Quartus II software 4.2.
	June 2004 v2.0	<ul style="list-style-type: none">● Updates to tables, figures.● New functionality for Quartus II software 4.1.
	Feb. 2004 v1.0	Initial release.
2	Dec. 2004 v2.1	<ul style="list-style-type: none">● Updated tables and figures.● New functionality for Quartus II software 4.2.
	June 2004 v2.0	<ul style="list-style-type: none">● Updates to tables and figures.● New functionality for Quartus II software 4.1.
	Feb. 2004 v1.0	Initial release.
3	Dec. 2004 v3.0	Reorganized chapter and updated information.
	Aug. 2004 v2.1	<ul style="list-style-type: none">● New functionality for Quartus II software 4.1 SP1.
	June 2004 v2.0	<ul style="list-style-type: none">● Updates to tables and figures.● New functionality for Quartus II software 4.1.
	Feb. 2004 v1.0	Initial release.

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 VHDL or Verilog HDL designs that target an Altera FPGA. This chapter provides step-by-step explanations of how to simulate your design in the ModelSim-Altera version or the ModelSim full 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 5.8d and using the ModelSim PE software version.



This chapter contains references to features available in the Altera Quartus® II software version 4.2. Visit the Altera web site, available at www.altera.com/quartus for information on the current Quartus II software version.

Background

The ModelSim-Altera software version 5.8d is included with your Altera software subscription, and can be licensed for the PC, Solaris, HP-UX, or Linux platforms to support either VHDL or Verilog hardware description language (HDL) simulation. The ModelSim-Altera tool supports VHDL or Verilog functional RTL, post-synthesis, and gate-level timing simulations for all Altera devices.

Table 1-1 describes the differences between the ModelSim-Modeltech and ModelSim-Altera versions.

Table 1-1. Comparison of ModelSim Versions (Part 1 of 2)

Product Feature	ModelSim SE	ModelSim PE	ModelSim-Altera
VHDL, Verilog, mixed-HDL support	optional	optional	Supports only single-HDL simulation
Complete HDL debugging environment	✓	✓	✓
Industry-standard scripting	✓	✓	✓
Flexible licensing	✓	optional	✓

Table 1–1. Comparison of ModelSim Versions (Part 2 of 2)

Product Feature	ModelSim SE	ModelSim PE	ModelSim-Altera
Verilog PLI (1) support. Interfaces Verilog designs to customer C code and third-party software	✓	✓	✓
VHDL FLI support. Interfaces VHDL designs to customer C code and third-party software	✓		
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 primitives (3 times faster), and register transfer level (RTL) acceleration (5 times faster)	✓		
Platform support	PC, UNIX, Linux	PC only	PC, UNIX, Linux

Note to Table 1–1:

- (1) See www.altera.com/products/software/pld/products/partners/eda-ms.html

Software Compatibility

Table 1–2 shows which ModelSim-Altera software version is compatible with the Quartus II software versions. ModelSim versions provided directly from Model Technology do not correspond to specific Quartus II software versions.



For help on ModelSim-Altera licensing set-up, see “Software Licensing & Licensing Set-Up” on page 1–29.

Table 1–2. Compatibility Between Software Versions

ModelSim-Altera Software	Quartus II Software (1)
ModelSim-Altera software version 5.8.e ModelSim-Altera software version 5.8.d	Quartus II software version 4.2
ModelSim-Altera software version 5.8c	Quartus II software version 4.1
ModelSim-Altera software version 5.7e	Quartus II software version 4.0

Note to Table 1–2:

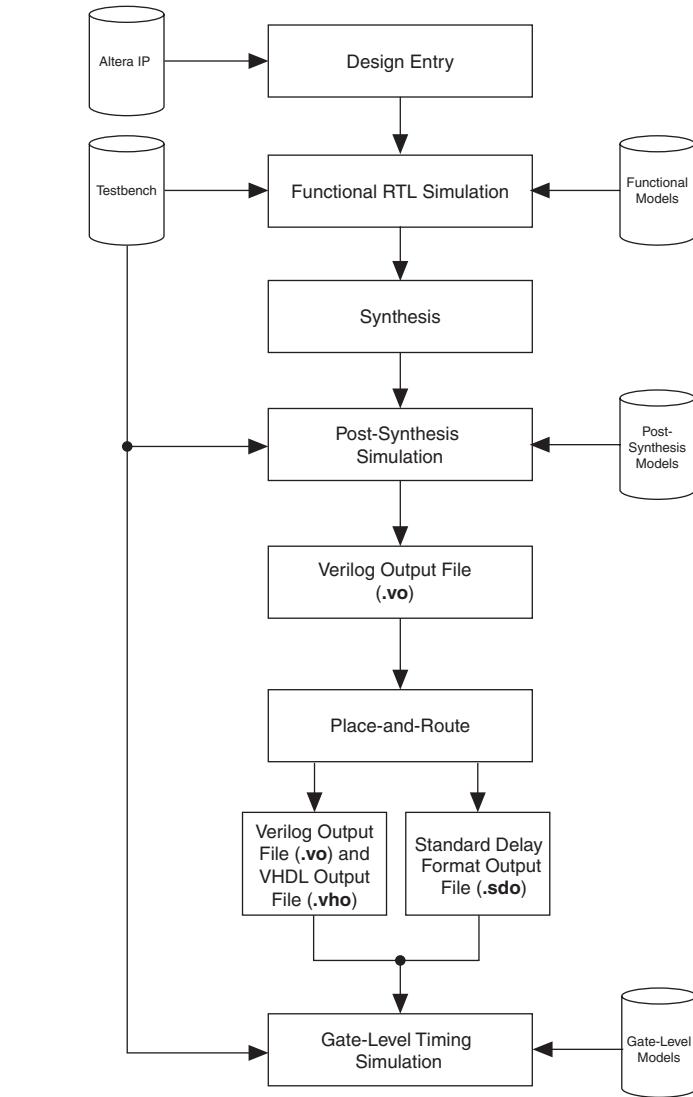
- (1) ModelSim-Altera precompiled libraries are updated with Quartus II release and service packs and are available for download on Altera’s web site.

Altera Design Flow with ModelSim-Altera Software

Figure 1–1 illustrates an Altera design flow using the ModelSim-Altera software or ModelSim full version:

- Functional RTL simulations
- Post-synthesis simulations
- Gate-level timing simulations

Figure 1–1. Altera Design Flow with ModelSim-Altera & Quartus II Software



Functional RTL Simulation

A functional RTL simulation is performed before a gate-level simulation or post-synthesis simulation. This 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 Model Technology ModelSim software versions for VHDL and Verilog HDL designs.

Functional RTL Simulation Libraries

LPM Simulation Models

To simulate designs containing LPM functions or MegaWizard® Plug-In Manager-generated functions, use the following Altera 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 1–3 shows the location of the simulation model files in the Quartus II software and the ModelSim-Altera software.

Table 1–3. Location of LPM Simulation Models	
Software	Location
Quartus II	<Quartus II installation directory>\eda\siml_lib\ (1)
ModelSim-Altera (PC)	<ModelSim-Altera installation directory>\altera\<HDL>\220model\ (2)
ModelSim-Altera (UNIX)	<ModelSim-Altera installation directory>/modeltech/altera/<HDL>/220model/ (1)

Notes to Table 1–3:

- (1) For Model Technology's ModelSim, use the files provided with the Quartus II software.
- (2) Compile **220pack.vhd** before **220model.vhd**.



For more information on LPM functions, see 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 1–4 shows the location of these files in the Quartus II software and the ModelSim-Altera software.

Table 1–4. Location of Altera Megafunction Simulation Models	
Software	Location
Quartus II	<code><Quartus II installation directory>\eda\sim_lib\ (1)</code>
ModelSim-Altera (PC)	<code><ModelSim-Altera installation directory>\altera\<HDL>\altera_mf\ (2), (3)</code>
ModelSim-Altera (UNIX)	<code><ModelSim-Altera installation directory>/modeltech/altera/\<HDL>/altera_mf/ (3)</code>

Notes to Table 1–4:

- (1) For Model Technology's ModelSim, use the files provided with the Quartus II software.
- (2) Compile `altera_mf_component.vhd` before `altera_mf.vhd`.
- (3) `<HDL>` can be either Verilog HDL or VHDL.

Simulating VHDL Designs

The following instructions helps you to perform a functional RTL simulation for VHDL designs in the ModelSim software.

The following steps assume you have already created a ModelSim project.

Create Simulation Libraries

Simulation libraries are required to simulate a design that contains an LPM function or an Altera megafunction. If you are using the Model Technology ModelSim software version, you must create the simulation libraries and link them to your design correctly.

 Creating a simulation library is not required if you are using the ModelSim-Altera software.

Create Simulation Libraries Using the ModelSim GUI

Perform the following steps to create simulation libraries:

1. In the ModelSim software, choose **New > Library** (File menu).
 2. In the **Create a New Library** dialog box select **a new Library and a logical linking 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. Enter the name of the newly created library in the **Library Name** box.
 4. Click **OK**.

Create Simulation Libraries Using the ModelSim Command Prompt

Type the following command lines at the ModelSim command prompt:

```
vlib altera_mf ←  
vmap altera_mf altera_mf ←  
vlib lpm ←  
vmap lpm lpm ←
```

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. Choose **Add to Project** (File menu) and select **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 **220model.vhd** model file should be compiled into the **lpm** library.
3. Select the simulation model file and choose **Properties** (View menu).
 4. Choose the correct library from the **Compile to Library** list.

5. Click **OK**.
6. Choose **Compile selected** (Compile menu).

Compile Simulation Models into Simulation Libraries Using the ModelSim Command Prompt:

Type the following command lines 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 ↵
```

Compile Testbench & Design Files into Work Library

To compile testbench and design files into a work library, choose **Compile All** (Compile menu) or click the **Compile All** toolbar icon.

Compile Testbench & Design Files into Work Library Using the ModelSim Command Prompt

Type the following command at the ModelSim command prompt:

```
vcom -work work <my_testbench.vhd> <my_design_files.vhd> ↵
```

 Resolve compile-time errors before proceeding to “[Loading the Design](#)” below.

Loading the Design

Perform the following steps to load a design:

1. Choose **Simulate** (Simulate menu).
2. Expand the work library in the **Simulate** dialog box.
3. Select the top-level design unit (your testbench). Click **OK** in the **Simulate** dialog box.

Loading the Design Using the ModelSim Command Prompt

Type the following command at the ModelSim command prompt:

```
vsim work.<my_testbench> -t ps ↵
```

Running the Simulation

Perform the following steps to run the simulation:

1. Choose **Signals** and **Wave** (View menu).
2. Drag signals to monitor from the **Signals** window and drop them into the **Wave** window.
3. At the ModelSim command prompt, type the following:

```
run <time period> ↵
```

Running the Simulation Using the ModelSim Command Prompt

Type the following command line at the ModelSim command prompt:

```
add wave /<signal name> ↵
run <time period> ↵
```

Simulating Verilog Designs

The following instructions provide step-by-step instructions on performing functional RTL simulation for Verilog designs in the ModelSim software.

The following steps assume you have already created a ModelSim project.

Create Simulation Libraries

Simulation libraries are needed to properly simulate a design that contains an LPM function or an Altera megafunction. If you are using the Model Technology ModelSim software version, you need to create the simulation libraries and correctly link them to your design.



Creating a simulation library is not required for the ModelSim-Altera software.

Create Simulation Libraries Using the ModelSim GUI

Perform the following steps to create simulation libraries:

1. Choose **New > Library** (File menu).
2. In the **Create a New Library** dialog box select **a new Library and a logical linking to it**.



The name of the libraries should be **altera_mf** (for Altera megafunctions) and **lpm** (for LPM and Megawizard-generated entities).

3. Enter the name of the newly created library in the **Library Name** box.
4. Click **OK**.

Create Simulation Libraries Using the ModelSim Command Prompt

Type the following command lines at the ModelSim command prompt:

```
vlib altera_mf <br>
vmap altera_mf altera_mf <br>
vlib lpm <br>
vmap lpm lpm <br>
```

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. Choose **Add to Project** (File menu) and select **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.v** should be compiled into the **altera_mf** library. Compile the **220model.v** into the **lpm** library.
3. Select the simulation model file and choose **Properties** (View menu).
 4. Choose the correct library from the **Compile to Library** list.
 5. Click **OK**.
 6. Choose **Compile selected** (Compile menu).

Compile Simulation Models into Simulation Libraries Using the ModelSim Command Prompt

Type the following command lines 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 ↵
```

Compile Testbench & Design Files into Work Library

To compile testbench and design files into a work library, choose **Compile All** (Compile menu) or click the **Compile All** toolbar icon.

Compile Testbench & Design Files into Work Library Using the ModelSim Command Prompt

Type the following command at the ModelSim command prompt:

```
vlog -work work <my_testbench.v> <my_design_files.v> ↵
```

 Resolve compile-time errors before proceeding to “[Loading the Design](#)” below.

Loading the Design

Perform the following steps to load a design:

1. Choose **Simulate** (Simulate menu).
2. Click the **Libraries** tab in the **Load Design** dialog box.
3. In the **Search Libraries** box, click **Add**.
4. Specify the location to the **lpm** or **altera_mf** simulation libraries.

 If you are using the ModelSim-Altera version, see [Table 1–3](#) and [Table 1–4](#) for the location of the precompiled simulation libraries. If you are using the ModelSim-Modeltech version, browse to the library that was created earlier.

5. In the **Load Design** dialog box, click the **Design** tab.
6. Expand the work library in the **Simulate** dialog box.
7. Select the top-level design unit (your testbench). Click **OK** in the **Simulate** dialog box.

Loading a Design Using the ModelSim Command Prompt

Type the following command at the ModelSim command prompt:

```
vsim -L <location of the altera_mf library> -L <location of the lpm  
library> work.<my_testbench> -t ps $\leftarrow$ 
```

Running the Simulation

Perform the following steps to run a simulation:

1. Choose **Signals** and **Wave** (View menu).
2. Drag signals to monitor from the **Signals** window and drop them into the **Wave** window.
3. At the ModelSim command prompt, type the following:

```
run <time period> $\leftarrow$ 
```

Running the Simulation Using the ModelSim Command Prompt

Type the following command line at the ModelSim command prompt:

```
add wave /<signal name> $\leftarrow$   
run <time period> $\leftarrow$ 
```

Verilog Functional RTL Simulation with Altera Memory Blocks

You can simulate your design containing complex memory blocks such as LPM_RAM_DP and ALTSYNCRAM using either ModelSim software version.

These memory blocks can be configured with initialization data using a hexadecimal (.hex) file or Memory Initialization File (.mif). The LPM_FILE parameter included in the HDL file generated by the MegaWizard Plug-In Manager points to the path of the HEX or MIF file that is used to initialize the memory block. You can create a HEX or MIF file with the Quartus II software.

Neither ModelSim software version can directly read the HEX or MIF file format. Therefore, to allow functional simulation of Altera memory blocks in the ModelSim software, you must perform the following steps:

1. Convert a HEX or MIF file to a RAM Initialization File (.rif).
2. Modify of the HDL file generated by the MegaWizard Plug-In Manager.
3. Compile the **nopl.v** file.

Converting a HEX File or MIF to a RIF

A RIF is an ASCII text file that you use with tools from EDA vendors. Create a RIF by converting an existing MIF or HEX file with the **Export** (File menu) command in the Quartus II software.

Modifying the MegaWizard Plug-In Manager-Generated File

You must modify the MegaWizard Plug-In Manager-generated file so that it includes the path to the newly created RIF. You must modify the LPM_FILE parameter. The following example shows the entry that you must change:

```
lpm_ram_dp_component.lpm_outdata = "UNREGISTERED",
lpm_ram_dp_component.lpm_file = "<path to RIF>"
lpm_ram_dp_component.use_eab = "ON",
```

Compiling nopli.v

The **nopli.v** file is included in the *<path to Quartus II installation>\eda\sim_lib* directory. This file contains the following definition:

```
`define NO_PLI 1
```

This definition instructs the ModelSim compile to read in the RIF.

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 has been 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 by choosing **Start > Start Analysis & Synthesis** (Processing menu).
2. Turn on the **Generate Netlist for Functional Simulation Only** by performing the following steps:
 - a. Choose **EDA Tool Settings** (Assignments menu).

- b. In the **Tool Name** list:
 - i. If you are using ModelSim-Altera, select **ModelSim-Altera (VHDL)** or **ModelSim-Altera (Verilog)**.
 - ii. If you are using Model Technology's ModelSim, select **ModelSim (Verilog)** or **ModelSim (VHDL)**.
 - c. Click **OK**.
3. Run the EDA Netlist Writer by choosing **Start > Start EDA Netlist Writer** (Processing menu).

During the EDA Netlist Writer stage, the Quartus II software produces a Verilog output (**.vo**) or VHDL output (**.vho**) file 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 `<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.

Create Simulation Libraries

Simulation libraries are needed to simulate a design that is mapped to post-synthesis primitives. If you are using the Model Technology ModelSim software version, you need to 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 are installed with the software.

Create Simulation Libraries Using the ModelSim GUI

Perform the following steps to create simulation libraries:

1. Choose **New Library** (File menu).
2. In the **Create a New Library** dialog box, select **a new Library and a logical linking to it**.
3. Enter in the name of the newly created library in the **Library Name** box.

4. Click **OK**.

Create Simulation Libraries Using the ModelSim Command Prompt

Type the following commands to create simulation libraries:

```
vlib <library name> ↵  
vmap <library name> <device family name> ↵
```

Compile Simulation Models into Simulation Libraries

This process is not required for the ModelSim-Altera version because a set of pre-compiled libraries are created when you install the software.

Compile Simulation Models into Simulation Libraries Using the ModelSim GUI

Perform the following steps to compile simulation models into simulation libraries:

1. Choose **Add to Project** (File menu), then select **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 select **Properties** (View menu).
4. Choose the correct library from the **Compile to Library** list.
5. Click **OK**.
6. Choose **Compile selected** (Compile menu).

Compile Simulation Models into Simulation Libraries Using the ModelSim Command Prompt

Type the following command lines at the ModelSim command prompt:

```
vcom -work <device family name> <Quartus II installation directory>  
/eda/sim_lib/<device family name>_components.vhd ↵
```

```
vcom -work <device family name> <Quartus II installation directory>  
/eda/sim_lib/<device family name>.vhd ↵
```

Compile Testbench & VHO into Work Library

1. To compile testbench and VHO files into a work library, choose **Compile All** (Compile menu) or click the **Compile All** toolbar icon.

Compile Testbench & VHO into Work Library Using ModelSim Command Prompt

Type the following command line at the ModelSim command prompt:

```
vcom -work work <my_testbench.vhd> <my_vhdl_output_file.vho>↔
```



Resolve any compilation errors before proceeding to *Loading the Design*.

Loading the Design

Perform the following steps to load a design:

1. Choose **Simulate** (Simulate menu).
2. In the **Library** list (**Design** tab), select the **work** library.
3. Expand the **work** library in the **Simulate** dialog box.
4. Select the top-level design unit (your testbench) and click **OK** in the **Simulate** dialog box.

Loading the Design Using the ModelSim Command Prompt

Type the following command line at the ModelSim command prompt:

```
vsim work.<my testbench> -t ps↔
```

Running the Simulation

Perform the following steps to run a simulation:

1. Choose **Signals** and **Wave** (View menu).
2. Drag signals to monitor from the **Signals** window and drop them into the **Wave** window.
3. At the ModelSim command prompt, type the following:

```
run <time period>↔
```

Running the Simulation Using the ModelSim Command Prompt

Type the following command lines at the ModelSim command prompt:

```
add wave /<signal name>←  
run <time period>←
```

Simulating Verilog Designs

The following provides step-by-step instructions on 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.



This process is not required for the ModelSim-Altera version because a set of pre-compiled libraries are created when you install the software. If you are using the Model Technology ModelSim software version, you need to 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. Choose **New Library** (File menu).
2. In the **Create a New Library** dialog box, select **a new library and a logical linking to it**.
3. Enter the name of the newly created library in the **Library Name**.
4. Click **OK**.

Create Simulation Libraries Using the ModelSim Command Prompt

Type the following command lines at the ModelSim command prompt:

```
vlib <library name>←  
vmap <library name><device family name>←
```

Compile Simulation Models into Simulation Libraries

This process is not required for the ModelSim-Altera version because a set of pre-compiled libraries are created when you install the software.

Compile Simulation Models into Simulation Libraries Using the ModelSim GUI

Perform the following steps to compile simulation models into simulation libraries:

1. Choose **Add to Project** (File menu) and 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 choose **Properties** (View menu).
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 line at the ModelSim command prompt:

```
vlog -work <device family name> <Quartus II installation directory>
/eda/sim_lib/<device family name>_atoms.v ↵
```

Compile Testbench & VO into Work Library

To compile testbench and VO files into a work library, choose **Compile All** (Compile menu) or click the **Compile All** toolbar icon.

Compile Testbench & VO into Work Library Using the ModelSim Command Prompt

Type the following command line at the ModelSim command prompt:

```
vlog -work work <my_testbench.v> <my_verilog_output_file.vo> ↵
```



Resolve any compilation errors before proceeding to *Loading the Design*.

Loading the Design

Perform the following steps to load a design:

1. Choose **Simulate** (Simulate menu).
2. In the **Load Design** dialog box, click the **Libraries** tab.
3. In the **Search Libraries** box, click **Add**.
4. Specify the location to the gate level simulation library.



If you are using the ModelSim-Altera version, refer to [Tables 1–5](#) and [1–6](#) for the location of the precompiled simulation libraries. If you are using the ModelSim-Modeltech version, browse to the library that was created earlier.

5. In the **Load Design** dialog box, click the **Design** tab.
6. Expand the work library in the **Simulate** dialog box.
7. Select the top-level design unit (your testbench) and click **OK** in the **Simulate** dialog box.

Loading the Design Using the ModelSim Command Prompt

Type the following command line at the ModelSim command prompt:

```
vsim -L <location of the gate level simulation library> work.<my_testbench>
-t ps ↵
```

Running the Simulation

Perform the following steps to run a simulation:

1. Choose **Signals** and **Wave** (View menu).
2. Drag signals to monitor from the **Signals** window and drop them into the **Wave** window.
3. At the ModelSim command prompt, type the following:

```
run <time period> ↵
```

Running the Simulation Using the ModelSim Command Prompt

Type the following command lines 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 Model Technology ModelSim software versions for VHDL and Verilog HDL designs.

Quartus II Software Output Files for use in the ModelSim-Altera Software

To perform gate-level timing simulation, the ModelSim-Altera software requires information on how the design was placed into device-specific architectural blocks. The Quartus II software provides this information in the form of a VO file for Verilog HDL designs and a VHO file for VHDL designs. The accompanying timing information is stored in the SDF file, which annotates the delay for the elements found in the VO or VHO output file.

To generate the VO or VHO files, perform the following steps:

1. Choose **EDA Tool Settings** (Assignments menu).
2. In the **Tool Name** list:
 - a. If you are using ModelSim-Altera, select **ModelSim-Altera (VHDL)** or **ModelSim-Altera (Verilog)**.
 - b. If you are using Model Technology's ModelSim, select **ModelSim (Verilog)** or **ModelSim (VHDL)**.
3. Click **OK**.
4. Compile the project.
5. The Quartus II output files are located in the *<full path to project>\simulation\ModelSim* directory.

Gate Level Simulation Libraries

Table 1–5 provides a description of the ModelSim-Altera precompiled device libraries.

Table 1–5. ModelSim-Altera Precompiled Device Libraries (Part 1 of 2)

Library	Description
stratixii	Precompiled library for Stratix II device designs
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 (altgxb Megafunction)
cycloneii	Precompiled library for Cyclone™ II device designs
cyclone	Precompiled library for Cyclone device designs
maxii	Precompiled library for MAX® II device designs

Table 1–5. ModelSim-Altera Precompiled Device Libraries (Part 2 of 2)

Library	Description
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 1–6 shows the location of the timing simulation libraries in the ModelSim-Altera software for Verilog HDL on PCs and UNIX workstations.

Table 1–6. Location of Timing Simulation Libraries for ModelSim-Altera for Verilog HDL

Library	Verilog HDL
stratixii	<ModelSim-Altera installation directory>\altera\verilog\stratixii\
stratix	<ModelSim-Altera installation directory>\altera\verilog\stratix\
stratixgx	<ModelSim-Altera installation directory>\altera\verilog\stratixgx\
stratixgx_gxb	<ModelSim-Altera installation directory>\altera\verilog\stratixgx_gxb\
cycloneii	<ModelSim-Altera installation directory>\altera\verilog\cycloneii\
cyclone	<ModelSim-Altera installation directory>\altera\verilog\cyclone\
maxii	<ModelSim-Altera installation directory>\altera\verilog\maxii\
max	<ModelSim-Altera installation directory>\altera\verilog\max\
apexii	<ModelSim-Altera installation directory>\altera\verilog\apexii\
apex20k	<ModelSim-Altera installation directory>\altera\verilog\apex20k\
apex20ke	<ModelSim-Altera installation directory>\altera\verilog\apex20ke\
mercury	<ModelSim-Altera installation directory>\altera\verilog\mercury\
flex10ke	<ModelSim-Altera installation directory>\altera\verilog\flex10ke\
flex6000	<ModelSim-Altera installation directory>\altera\verilog\flex6000\

Table 1–7 shows the location of the timing simulation libraries in the ModelSim-Altera software for VHDL on PCs or UNIX workstations.

Library	VHDL
stratixii	<ModelSim-Altera installation directory>\altera\vhdl\stratixii\
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\
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\

If you are using the ModelSim-Modeltech version for your timing simulation, libraries are available in the Quartus II software at the following directory: <Quartus II installation directory>\eda\sim_lib\. Model Technology ModelSim software users must use the files provided with the Quartus II software.

Simulating VHDL Designs

The following 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 see “Altera Design Flow with ModelSim-Altera Software” on page 1–3.

Create Simulation Libraries

If you are using the Model Technology ModelSim software version, create the gate-level simulation libraries and correctly link them to your design.



This process is not required for the ModelSim-Altera version because a set of pre-compiled libraries are created when you install the software.

Create Simulation Libraries Using the ModelSim GUI

Perform the following steps to create simulation libraries using the ModelSim GUI:

1. Choose **New Library** (File menu).
2. In the **Create a New Library** dialog box, select **a new Library and a logical linking to it**.
3. Type in the name of the newly created library in the **Library Name** box.
4. Click **OK**.

Create Simulation Libraries Using the ModelSim Command Prompt

Type the following command lines at the ModelSim command prompt:

```
vlib <library name> ↵  
vmap <library name> <device family name> ↵
```

Compile Simulation Models into Simulation Libraries

This process is not required for the ModelSim-Altera version because a set of pre-compiled libraries are created when you install the software.

Compile Simulation Models into Simulation Libraries Using the ModelSim GUI

Perform the following steps to compile simulation models into simulation libraries:

1. Choose **Add to Project** (File menu), then select **Existing File**.
2. Browse to the *<Quartus II installation directory>/eda/sim_lib* and add the necessary gate level simulation files to your project.
3. Select the simulation model file and select **Properties** (View menu).
4. Choose the correct library from the **Compile to Library** list.

5. Click **OK**.
6. Choose **Compile selected** (Compile menu).

Compile Simulation Models into Simulation Libraries Using the ModelSim Command Prompt

Type the following command lines at the ModelSim command prompt:

```
vcom -work <device family name> <Quartus II installation directory>
/eda/sim_lib /<device family name>_components.vhd ↵
```

```
vcom -work <device family name> <Quartus II installation directory>
/eda/sim_lib /<device family name>.vhd ↵
```

Compile Testbench & VHO into Work Library

To compile testbench and VHO files into a work library, choose **Compile All** (Compile menu) or click the **Compile All** toolbar icon.

Compile Testbench & VHO into Work Library Using the ModelSim Command Prompt

Type the following command line at the ModelSim command prompt:

```
vcom -work work <my_testbench.vhd> <my_vhdl_output_file.vho> ↵
```

 Resolve any compilation errors before proceeding to *Loading the Design*.

Loading the Design

Perform the following steps to load a design:

1. Choose **Simulate** (Simulate menu).
2. Click the **SDF** tab and click **Add**.
3. Specify the location of the SDF file and click **OK**.
4. In the **Library** list (**Design** tab), select the **work** library.
5. Expand the **work** library in the **Simulate** dialog box.
6. Select the top-level design unit (your testbench) and click **OK** in the **Simulate** dialog box.

Loading a Design Using the ModelSim Command Prompt

Type the following command line at the ModelSim command prompt:

```
vsim -sdftyp work.<my_testbench> -t ps
```

Running the Simulation

Perform the following steps to run the simulation:

1. Choose **Signals** and **Wave** (View menu).
2. Drag signals to monitor from the **Signals** window and drop them into the **Wave** window.
3. At the ModelSim command prompt, type the following:

```
run <time period>
```

Running a Simulation Using the ModelSim Command Prompt

Type the following command lines at the ModelSim command prompt:

```
add wave /<signal name>
run <time period>
```

Simulating Verilog Designs

The following provides step-by-step instructions on performing gate-level timing simulation for Verilog HDL designs in the ModelSim-Altera software.



The following steps assume you have already created a ModelSim project. For additional information see “[Altera Design Flow with ModelSim-Altera Software](#)” on page 1–3.

Create Simulation Libraries

If you are using the Model Technology ModelSim software version, you need to create the simulation libraries and correctly link them to your design.



This process is not required for the ModelSim-Altera version because a set of pre-compiled libraries are created when you install the software.

Create Simulation Libraries Using the ModelSim GUI

Perform the following steps to create simulation libraries:

1. Choose **New Library** (File menu).
2. In the **Create a New Library** dialog box, select **a new library and a logical linking to it**.
3. Enter the name of the newly created library in the **Library Name**.
4. Click **OK**.

Create Simulation Libraries Using the ModelSim Command Prompt

Type the following command lines at the ModelSim command prompt:

```
vlib <library name> ↵  
vmap <library name> <device family name> ↵
```

Compile Simulation Models into Simulation Libraries

This process is not required for the ModelSim-Altera version because a set of pre-compiled libraries are created when you install the software.

Compile Simulation Models into Simulation Libraries Using the ModelSim GUI

Perform the following steps to compile simulation models into simulation libraries:

1. Choose **Add to Project** (File menu) and select **Existing File**.
Browse to the *<Quartus II installation directory>/eda/sim_lib* and add the necessary simulation model files to your project.
2. Select the simulation model file and select **Properties** (View menu).
3. Choose the correct library from the **Compile to Library** list.
4. Click **OK**.
5. Choose **Compile selected** (Compile menu).

Compile Simulation Models into Simulation Libraries Using the ModelSim Command Prompt

Type the following command line at the ModelSim command prompt:

```
vlog -work <device family name> <Quartus II installation directory>  
/eda/sim_lib /<device family name>_atoms.v ↵
```

Compile Testbench & VO into Work Library

To compile testbench and VO files into a work library, choose **Compile All** (Compile menu) or click the **Compile All** toolbar icon.

Compile Testbench & VO into Work Libraries Using the ModelSim Command Prompt

Type the following command line at the ModelSim command prompt:

```
vlog -work work <my_testbench.v> <my_verilog_output_file.vo>↔
```



Resolve any compilation errors before proceeding to *Loading the Design*.

Loading the Design

Perform the following steps to load a design:

1. Choose **Simulate** (Simulate menu).
 2. In the **Load Design** dialog box, click the **Libraries** tab.
 3. In the **Search Libraries** box, click **Add**.
 4. Specify the location to the gate-level simulation library.
- If you are using the ModelSim-Altera version, refer to [Tables 1–5](#) and [1–6](#) for the location of the precompiled simulation libraries. If you are using the ModelSim-Modeltech version, browse to the library that was created earlier.
5. In the **Load Design** dialog box, click the **Design** tab.
 6. Expand the work library in the **Simulate** dialog box.
 7. Select the top-level design unit (your testbench) and click **OK** in the **Simulate** dialog box.

Loading the Design Using the ModelSim Command Prompt

Type the following command line at the ModelSim command prompt:

```
vsim -L <location of the gate level simulation library>
      -work.<my_testbench> -t ps↔
```

Running the Simulation

Perform the following steps to run the simulation:

1. Choose **Signals** and **Wave** (View menu).
2. Drag signals to monitor from the **Signals** window and drop them into the **Wave** window.
3. At the prompt, type the following:

```
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>↔
```

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.

To run an EDA simulation or timing analysis tool automatically after a compilation in the Quartus II software:

1. Choose **EDA Tool Settings** (Assignments menu) and set the **Simulation Tool Name** to one of the following:

- ModelSim (VHDL)
- ModelSim (Verilog)
- ModelSim-Altera (VHDL)
- ModelSim-Altera (Verilog)



Make sure you turn on **Run this tool automatically after compilation** in the **Simulation** page under **EDA Tool Settings** in the **Settings** dialog box (Assignments menu).

2. Compile the design.

The Quartus II software creates a simulation work directory, compiles the appropriate design files and simulation libraries, and launches the EDA simulation tool.

Scripting Support



To run ModelSim automatically from the Quartus II software using the NativeLink feature on a UNIX workstation, you must add the following environment variable to your `.cshrc` file:

```
QUARTUS_INI_PATH <ModelSim installation directory>
```

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, see the *Tcl Scripting* chapter in the *Quartus II Handbook Volume 2*. For more information about command line scripting, see the *Command-Line Scripting* chapter in the *Quartus II Handbook Volume 2*. For detailed information about scripting command options, see the **Qhelp** utility.

Type this command to start the Qhelp utility:

```
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_OUTPUT_DATA_FORMAT "VERILOG"
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 VCS simulator; specify vhdl or verilog for the format:

```
quartus_eda <project name> --simulation=on --
format=<format> --tool=Modelsim --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 the command-line prompt.

Tcl Commands

Use the following Tcl commands:

```
set_global_assignment -name EDA_OUTPUT_DATA_FORMAT "VERILOG"  
set_global_assignment -name EDA_SIMULATION_TOOL "Modelsim (Verilog)"
```

Command Prompt

Use the following command to generate a simulation output file for the VCS simulator. Specify vhdl or verilog for the format.

```
quartus_eda<project name> --simulation=on  
--format=<format> --tool=Modelsim ↵
```

Software Licensing & Licensing Set-Up

License the ModelSim-Altera software through a parallel port software guard (T-guard), FIXEDDPC license, or a network FLOATNET or FLOATPC license. Each Altera software subscription includes a license to either VHDL or Verilog HDL. Network licenses with multiple users may have their licenses split between VHDL and Verilog HDL in any ratio.



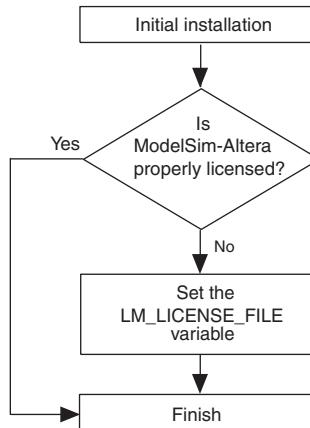
The USB software guard is not supported.

Obtain licenses for ModelSim-Altera software from the Altera web site at www.altera.com. Get licensing information for Model Technology's ModelSim directly from Model Technology. See [Figure 1-2](#) 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 1–2. 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.

Introduction

This chapter is an overview on 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 document contains references to features available in the Altera Quartus® II software version 4.2. For more information on the Quartus II software version 4.2, go to the Altera web site at www.altera.com.

Software Requirements

To properly simulate your design using VCS, you must set up the Altera libraries. These libraries are installed with the Quartus II software.

Table 2–1 shows the compatibility between versions of the Quartus II software and the Synopsys VCS software.

Table 2–1. Supported Quartus II & VCS Software Version Compatibility

Synopsys	Altera
VCS software version 7.1.1	Quartus II software version 4.2
VCS software version 7.1.1	Quartus II software version 4.1
VCS software version 7.0.1	Quartus II software version 4.0



See the *Quartus II Installation & Licensing for PCs* or the *Quartus II Installation & Licensing for UNIX and Linux Workstation* manuals for more information on installing the software and the directories created during the Quartus II software installation.

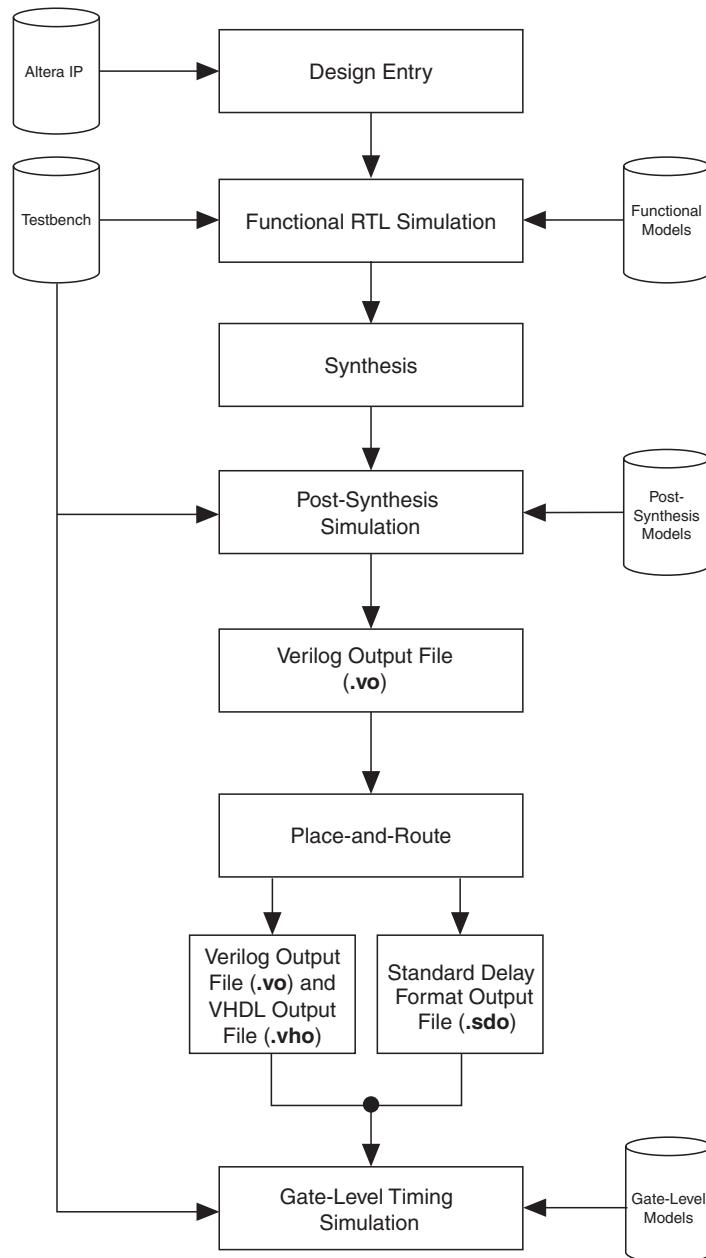
Using VCS in the Quartus II Design Flow

You can perform the following types of simulations using VCS:

- Functional RTL simulations
- Post-synthesis simulations
- Gate-level timing simulations

Figure 2–1 shows the VCS and Quartus II software design flow.

Figure 2–1. Altera Design Flow with the VCS & Quartus II Software



Functional RTL Simulations

Functional RTL simulations verify the functionality of the design before synthesis and before place-and-route. 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 for place-and-route.

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 2–2** summarizes the Verilog library files that are required to compile LPM functions and Altera megafunctions.

Table 2–2. Altera Verilog Functional/Behavioral Simulation Library Files	
Library File	Description
altera_mf.v	Libraries that contain simulation models for Altera megafunctions.
stratixgx_mf.v (1)	Libraries that contain simulation models for Stratix® GX devices.
220model.v	Libraries that contain simulation models for Altera LPM functions version 2.2.0.
sgate.v	Libraries that contain simulation models for Altera IP.

Note to Table 2–2:

- (1) When performing a functional RTL simulation of Stratix GX design you need to compile the **stratixgx_mf.v**, **sgate.v**, and **220model.v** files.

The files in **Table 2–2** are installed with a Quartus II installation. These files are found in the `<path to Quartus II installation>\eda\sim_lib` directory.

The following VCS command describes the command-line syntax to perform a functional RTL simulation with a pre-existing library:

```
vcs -R <test bench>.v <design name>.v
      -v <Altera library file>.v
```

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 (**.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 HEX file or MIF that is used to initialize the memory block. You can create a HEX file or MIF with the Quartus II software.

However, the VCS software cannot read the HEX or MIF file format. Therefore, in order to perform functional simulation of Altera memory blocks in the VCS software, you must perform the following steps:

1. Convert a HEX file or MIF to a RAM Initialization File (**.rif**)
2. Modify the file generated by the MegaWizard Plug-In Manager
3. Compile the **nopl.v** file



For more information on creating a MIF, see Quartus II Help.

Converting a HEX File or MIF to a RIF

A RIF is an ASCII text file that you can use with tools from electronic design automation (EDA) vendors. You can create a RIF by converting an existing MIF or HEX file using the **Export Current File As** command in the Quartus II software. This option is available in the File menu while the Quartus II Memory Editor is open.

Modifying the File Generated By the MegaWizard Plug-In Manager

You must modify the `lpm_file` parameter in the megafunction wizard-generated file so it includes the path to the newly created RIF. The following example shows the entry that you must change where `<path to RIF>` is the path to the RIF:

```
lpm_ram_dp_component.lpm_outdata = "UNREGISTERED",
lpm_ram_dp_component.lpm_file = "<path to RIF>",
lpm_ram_dp_component.use_eab = "ON",
```

Compiling nopl.v

The **nopl.v** file is included in the `<path to Quartus II installation>\eda\sim_lib` directory. This file contains the following definition:

```
`define NO_PLI 1
```

This definition instructs the VCS compile to read in the RIF.

The following VCS command simulates a design that includes Altera RAM blocks that require memory initialization at the Unix prompt:

```
vcs -R <path to Quartus installation>\eda  
  \sim_lib\noplib.v <test bench>.v  
  <design name>.v -v <Altera library file>.v ↵
```

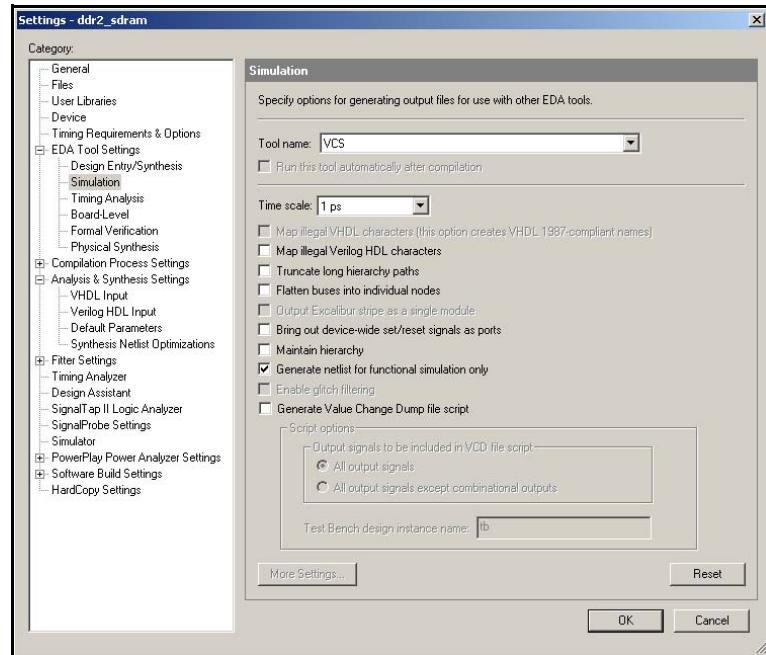
Post-Synthesis Simulation

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 VCS. 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 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 by choosing **Start** > **Start Analysis & Synthesis** (Processing menu).
2. Choose **Settings** (Assignments menu). In the **Category** list, select **EDA Tool Settings** (expand if necessary) > **Simulation**. In the **Tool name** list, choose **VCS** as shown in [Figure 2–2](#).
3. Turn on the **Generate netlist for functional simulation only**:
4. Click **OK**.

Figure 2–2. Setting the Tool Name to VCS in the Settings Window

5. Choose Start > Start EDA Netlist Writer (Processing menu).

During the EDA Netlist Writer stage, the Quartus II software produces a Verilog output (**.vo**) file 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 *<project folder>/simulation/VCS* directory. This netlist, along with the device family library listed in [Table 2–3 on page 2–8](#), can be used to perform a post-synthesis simulation in VCS.

The following VCS commands describe the command-line syntax used to perform a post-synthesis simulation with the appropriate device family library listed in [Table 2–3 on page 2–8](#):

```
vcs -R <testbench.v> <post synthesis netlist.vo> -v <altera device
family library.v>
```

Gate-Level Timing Simulation

A gate-level timing simulation verifies the functionality of the design after place-and-route has been performed. You can create a post-place-and-route netlist in the Quartus II software and use this netlist to perform a gate-level timing simulation in VCS.

Generating a Gate-Level Timing Simulation Netlist the in Quartus II Software

The following steps describe the process of generating a gate-level timing simulation netlist in the Quartus II software:

1. Start Compilation:

Choose **Start > Start Compilation** (Processing menu).

2. When the compilation has completed successfully, set the **Tool name** to **VCS** by choosing **Settings** (Assignments menu). In the **Category** list, select **EDA Tool Settings** (expand if necessary) > **Simulation**. In the **Simulation** section of the window, select **VCS** in the **Tool name** list, as shown in [Figure 2–2](#).
3. Run the EDA Netlist Writer by choosing **Start > Start EDA Netlist Writer** (Processing menu).

During the EDA Netlist Writer stage, the Quartus II software produces VO netlist file and a Standard Delay Output (.sdo) file used for a gate-level timing simulation in the VCS software. This netlist file is mapped to architecture-specific primitives. The SDO file contains timing delay information for each architecture primitive. Together, these files provide an accurate simulation of the design in the Altera FPGA architecture.

The resulting files are located in the `<project folder>/simulation/VCS` directory. These files, along with the device family library listed in [Table 2–3](#), are used to perform a gate-level timing simulation in VCS.

The following command-line syntax is used to perform gate-level timing with the device family library:

```
vcs -R <testbench.v> <gate-level timing netlist.vo> -v <altera device family library.v>
```

Table 2–3. Altera Gate-Level Simulation Libraries

Library Files	Description
stratixii_atoms.v	Atom libraries for Stratix II designs
stratix_atoms.v	Atom libraries for Stratix designs
stratixgx_atoms.v stratixgx_hssi_atoms.v	Atom libraries for Stratix GX designs
hc_stratix_atoms.v	Atom libraries for HardCopy® Stratix 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

Transport Delays

By default, VCS filters out all pulses that are shorter than the propagation delay between primitives. Turning on the transport delay options in VCS 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 primitive. For this option to work you must also include the **+pulse_int_e/number** and **+pulse_int_r/number** options.



For more information on either of these options, refer to the VCS *User Guide* installed with the VCS software.

The following VCS command describes the command-line syntax to perform a post-synthesis simulation with the device family library:

```
vcs -R <testbench.v> <gate-level netlist.vo> -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
```

Common VCS Compiler Options

The VCS software has a set of options that help you simulate your design. [Table 2–4](#) lists some of the options that are available.

Table 2–4. VCS Compiler Options

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 library file (i.e., 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 library directory. The VCS software looks for library files in this folder that contain module definitions that are instantiated in the source code.
+compsdf	Indicates that the VCS compiler includes the back-annotated SDF file in the compilation.
+cli	After successful completion of compilation, Command Line Interface (CLI) Mode is entered.
+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 on any VCS option, refer to the *VCS User Guide*.

Using VirSim: The VCS Graphical Interface

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 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 on using VirSim, see the *VirSim User Manual* included in the VCS installation.

VCS Debugging Support VCS 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 CLI can be used to halt simulations at user-defined break points, force registers with values, and display values of registers. To enable the non-graphical capability, you must use the `+cli` run-time option. To use the VCS CLI to debug your Altera FPGA design, use 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 on the `+cli` options, see the *VCS User Guide* included in the VCS installation.

Using PLI Routines with the VCS Software

The VCS software can interface your custom-defined C code with Verilog 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 language.

Preparing & Linking C Programs to Verilog 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 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 by using 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 source code. You can create the TAB file using a standard text editor. The following is an example of an entry in the TAB file:

```
$my_custom_function call=my_custom_function acc+=rw* ↵
```

The Verilog 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> ↵
```

Scripting Support



You can run procedures and create settings described in this chapter in a tool command language (Tcl) script. You can also run some procedures at a command prompt.

For more information about Tcl scripting, see the *Tcl Scripting* chapter in Volume 2 of the *Quartus II Handbook*.

For more information about command-line scripting, see the *Command-Line Scripting* chapter in Volume 2 of the *Quartus II Handbook*.

For detailed information about scripting command options, see the **Qhelp** utility.

Type this command to start the Qhelp utility:

```
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_OUTPUT_DATA_FORMAT "VERILOG"  
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 simulator; specify vhdl or verilog 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

```
set_global_assignment -name EDA_OUTPUT_DATA_FORMAT "VERILOG"  
set_global_assignment -name EDA_SIMULATION_TOOL "VCS"
```

Command Prompt

Use the following command to generate a simulation output file for the VCS simulator. Specify vhdl or verilog 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 make this simulation flow an ideal method for fully verifying an FPGA design.

qii53003-3.0

Introduction

This chapter is a getting started guide to using the Cadence Incisive verification platform in Altera® FPGA design flows. The Incisive verification platform 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 document contains references to features available in the Altera Quartus® II version 4.2 software. See the Altera web site at www.altera.com for information on the Quartus II version 4.2 software.

Software Requirements

You must first install the Quartus II software before using it with the Cadence Incisive platform. The Quartus II/Cadence interface is installed automatically when you install the Quartus II software on your computer.

Table 3–1 shows the Cadence NC simulator versions compatible with specific Quartus II software versions.

Table 3–1. Compatibility Between Software Versions

Quartus II Software	Cadence NC Simulators (UNIX)	Cadence NC Simulators (PC)	Cadence NC Simulators (Linux)
Version 4.0	Version 5.0 s005	Version 5.0 s006	Version 5.0 p001
Version 4.1	Version 5.1 s012	Version 5.1 s010	Version 5.0 p001
Version 4.2	Version 5.1 s017	Version 5.1 s017	Version 5.1 s017



See the *Quartus II Installation & Licensing for PCs* or the *Quartus II Installation & Licensing for UNIX and Linux Workstations* manual for more information on installing the software and the directories that are created during the Quartus II software installation.

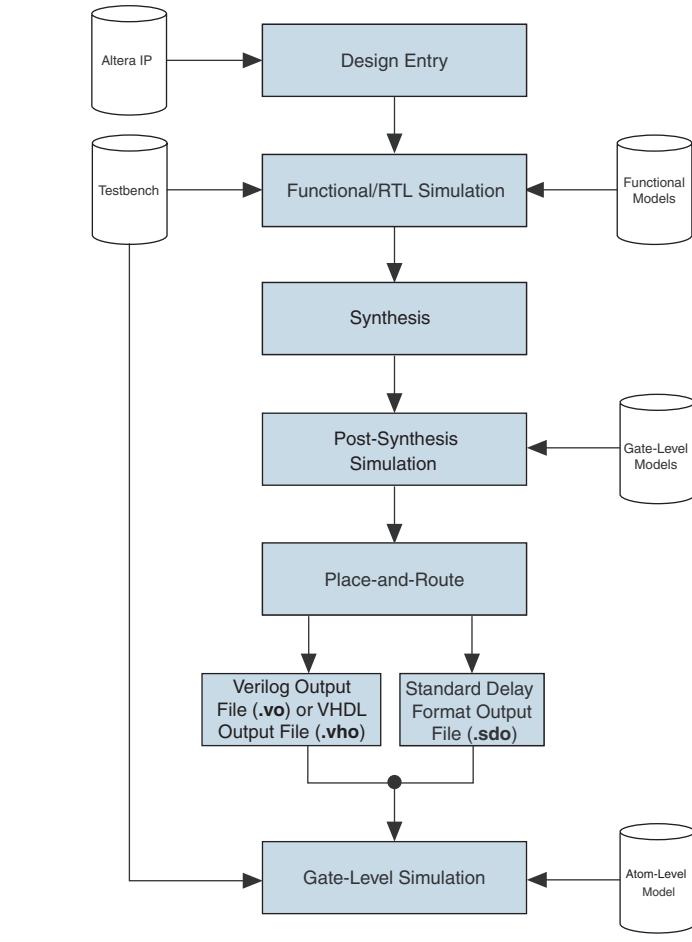
Simulation Flow Overview

The Incisive platform software supports the following simulation flows:

- Functional/register transfer level (RTL) simulation
- Post-synthesis simulation
- Gate-level timing simulation

Figure 3–1 shows the Quartus II and Cadence design flow.

Figure 3–1. Quartus II Design Flow With Cadence NC Simulators



Functional/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. See “[Functional/RTL Simulation](#)” on page 3–5 for more information on 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. See “[Post-Synthesis Simulation](#)” on page 3–20 for more information on 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 format (SDO) 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. See “[Gate-Level Timing Simulation](#)” on page 3–22 for more information on how to perform this simulation.

Operation Modes

In the NC simulators, you can use either the graphical user interface (GUI) mode or the command-line mode to simulate your design.

You launch 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 3–2](#).



This chapter describes how to perform simulation using both the GUI and the command-line.

Table 3–2. 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 & 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/RTL Simulation” on page 3–5, “Post-Synthesis Simulation” on page 3–20, and “Gate-Level Timing Simulation” on page 3–22.

1. Set up your working environment (UNIX only).

For UNIX workstations, you must set several environment variables 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, for 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 by choosing **Verilog** or **VHDL Compiler** (Tools menu) 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 choose **Elaborator** (Tools menu) 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/RTL Simulation

The following sections provide detailed instructions for performing functional/RTL simulation using the Quartus II software and the Incisive platform tools.

Set Up Your Environment

This section describes how to set up your working environment for the Quartus II software and Incisive platform interface. The information presented here assumes that you are using the C shell and that your Quartus II system directory is **/usr/quartus**. If not, you must use the appropriate syntax and procedures to set environment variables for your shell.

1. Add the following environment variables to your **.cshrc** file:

```
setenv QUARTUS_ROOTDIR /usr/quartus
setenv CDS_INST_DIR <NC installation directory>
```



The path notation (slashes) are for UNIX workstations. For PCs, reverse the slashes.

2. Add the `$CDS_INST_DIR/tools/bin` directory to the PATH environment variable in your `.cshrc` file. Make sure this path is placed before the Cadence hierarchy path.
3. Add `/usr/dt/lib` and `/usr/ucb/lib` to the `LD_LIBRARY_PATH` environment variable in your `.cshrc` file.
4. Source your `.cshrc` file by typing `source .cshrc ↵` at the command prompt.

The following sample provides an example for setting these environment variables.

```
setenv QUARTUS_ROOTDIR /usr/quartus
setenv CDS_INST_DIR <NC installation directory>
setenv PATH ${PATH}:<NC installation directory>/tools.sun4v/bin:/
setenv LD_LIBRARY_PATH
/usr/ucb/lib:/usr/lib:/usr/dt/lib:/usr/bin/X11:
<NC installation directory> /tools.sun4v/lib:$LD_LIBRARY_PATH
setenv QUARTUS_INIT_PATH <NC installation directory>/tools.sun4v/bin
```

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 launch 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 (for example, technology or cell libraries) and a user `cds.lib` file. The following sections describe how to create and edit a `cds.lib` file, including the following:

- Basic library setup
- LPM function and Altera megafunction 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 to an included **cds.lib** file in quotation marks if there are spaces in any directory names.

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, Synopsys libraries, and so on.

Create a cds.lib File in Command-Line Mode

To edit a **cds.lib** file from the command line, 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 <physical directory> ↵
```

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

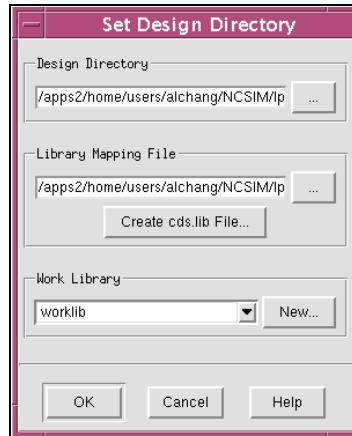
To create a **cds.lib** file using the GUI, perform the following steps:

1. Type `nclaunch ↵` at the command line to launch the GUI.
2. If the NCLaunch window is not in multiple step mode, choose **Switch to Multiple Step** (File menu).
3. Change your design directory by choosing **Set Design Directory** (File menu).

The Set Design Directory window opens, as shown in [Figure 3-2](#).

4. Click the **Browse** button (...) to navigate to your project directory.
5. Click **Create cds.lib File** and in the **New cds.lib File** dialog box, choose the libraries 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 3-2](#) shows an example using the directory name `worklib`.

Figure 3–2. Creating a Work Directory in GUI Mode



9. Click **OK**.



You can edit the **cds.lib** file by right-clicking the **cds.lib** filename in the right side of the window and choosing **Edit** from the right-button pop-up menu.

LPM Function & Altera Megafunction Libraries

Altera provides behavioral descriptions for LPM functions and Altera-specific megafunctions. 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. If your design uses LPM functions or Altera megafunctions, you must set up resource libraries so that you can simulate your design in the Incisive simulator.



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 the section “[Simulating a Design With Memory](#)” on [page 3–11](#).

Altera provides megafunction behavioral descriptions in the files shown in [Table 3–3](#). These library files are located in the:

<*Quartus II installation*>/**eda/sim_lib** directory.

For more information on LPM functions and Altera megafunctions, see the *Quartus II Help*.

Table 3–3. Megafunction Behavioral Description Files		
Megafunction	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
altgxb	stratixgx_mf.v (3)	stratixgx_mf.vhd (3) stratixgx_mf_components.vhd (3)
IP functional simulation model	sgate.v	sgate.vhd sgate_pack.vhd

Notes to Table 3–3:

- (1) Use this model with VHDL 93.
- (2) Use this model with VHDL 87.
- (3) The altgxb library files require the LPM and SGATE libraries.

To set up a library for LPM functions, create a new directory and add the following line to your **cds.lib** file:

```
DEFINE lpm <path>/<directory name>
```

To set up a library for Altera megafunctions, add the following line to your **cds.lib** file:

```
DEFINE altera_mf <path>/<directory name>
```

Simulating a Design With Memory

Many Altera functional models (220model.v and altera_mf.v) use a memory file, which is a Hexadecimal (Intel-Format) File (.hex) or a Memory Initialization File (.mif). However, Incisive simulators cannot read a HEX or MIF. Perform the following steps to convert these files into a format the tools can read:

1. In the Quartus II software, follow these steps to convert your HEX or MIF into a RAM Initialization File (.rif):



You can also use the **hex2rif.exe** and **mif2rif.exe** programs, located in the *<Quartus II installation directory>/bin* directory, to convert the files at the command line. Use the -? option for information on how to use these programs.

- a. Open the HEX or MIF file.
 - b. Choose **Export** (File menu).
 - c. In the **Save as type** list, choose **RAM Initialization File (.rif)**.
 - d. Click **Export**.
2. Using a text editor, modify the `lpm_file` parameter to point to the RIF file in the megafunction file.



Alternatively, you can run the MegaWizard Plug-In Manager and point to the RIF file as the memory initialization file.

The following examples show the entry that you must change:

- **Example 1:** dual_port_ram
`lpm_ram_dp_component.lpm_file = "<path to RIF>"`
- **Example 2:** altsyncram
`altsyncram_component.init_file = "<path to RIF>"`

Compile Source Code & 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 specify a work directory before compiling.

- Verilog HDL
`ncvlog <options> -work <library name> <design files>` ↵
- VHDL
`ncvhdl <options> -work <library name> <design files>` ↵

If your design uses LPM or Altera megafunctions, 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` ↵

If 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 ↵
```

Or use the NO_PLI command during compilation:

```
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` ↵

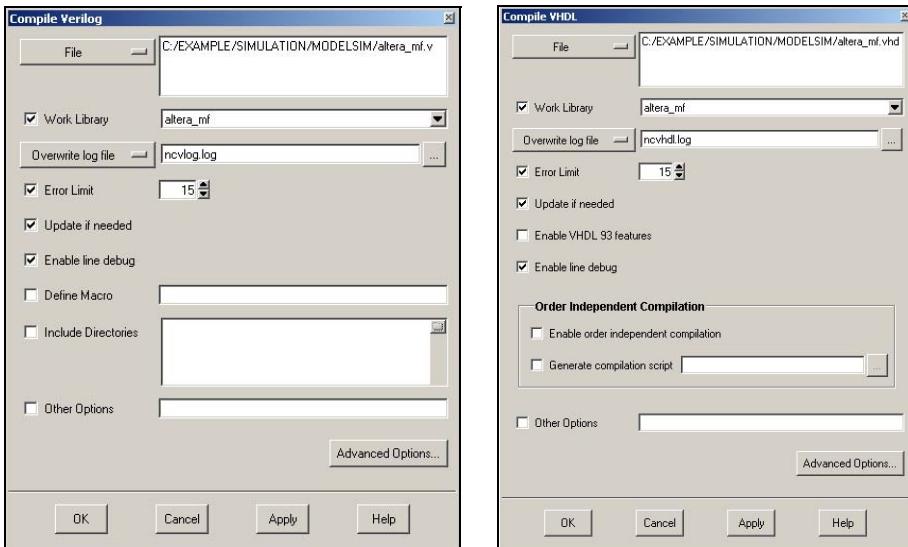
Compilation in GUI Mode

To compile using the NCLaunch GUI, perform the following steps:

1. Right-click a library filename in the NCLaunch window.
2. Choose NCVlog (Verilog HDL) or NCVhdl (VHDL).

The **Compile Verilog** or **Compile VHDL** dialog boxes open, as shown in [Figure 3–3](#). Alternatively, you can choose NCVlog or NCvhdl (Tools menu).

Figure 3–3. Compiling Verilog HDL & VHDL Files



3. Select the files 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>] ↵
```

For example: `ncelab worklib.lpm_ram_dp_test:entity` ↵

You can set your simulation timescale using the **-TIMESCALE <arguments>** option. For example:

```
ncelab -TIMESCALE 1ps/1ps worklib.lpm_ram_dp_test:entity ↵
```



If you specified a timescale of 1 ps in the Verilog testbench, the **TIMESCALE** option is not necessary. Using a ps resolution ensures the correct simulation of your design.

To view the elements in your library and the views that are available, use the **ncls** program. For example, the following command displays all of the cells and their views in your current **worklib** directory.

```
ncls -library worklib ↵
```



For more information on the **ncls** program, see the Cadence NC-Verilog Simulator Help or Cadence NC-VHDL Simulator Help.



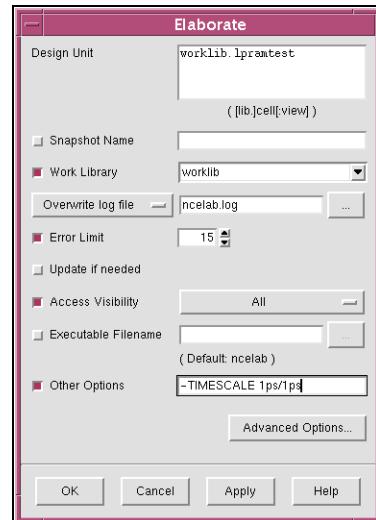
If you are running the NC-Verilog simulator using multistep invocation, run **ncelab** with command-line options as shown above.

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.
4. Choose **NCElab**. The **Elaborate** dialog box (Figure 3–4) opens. Optionally, you can choose **Elaborator** from the Tools menu.
5. Set the simulation timescale with the command **-TIMESCALE 1ps/1ps** in the **Other Options** box. See Figure 3–4.

Figure 3–4. Elaborating the Design



- Click **OK** in the **Elaborate** dialog box to begin elaboration. The dialog box closes and returns you to **NCLaunch**.



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, do steps 4 through 10 in the section “[Compiling the Standard Delay Output File \(VHDL Only\) in GUI Mode](#)” on page 3–26.

Add Signals to View

You 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 this database by adding commands to your code. Alternately, you can create a Value Change Dump File (.vcd) to store the simulation history.



For more information on using a VCD, see the Cadence NC-Sim user manual included in the installation.

Adding Signals in Command-Line Mode

To create an SHM database, specify the system tasks, described in [Table 3–4](#), in your Verilog HDL code.



For VHDL, you can use the Tcl command interface or C function calls to add signals to a database. See the Cadence documentation included in the installation for details.

Table 3–4. SHM Database System Tasks

System Task	Description
<code>\$shm_open("<filename>.shm") ;</code>	Opens a database. If you do not specify a filename, the default waves.shm database is opened. If a database with the specified name does not exist, it is created for you.
<code>\$shm_probe("[A S C]");</code>	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.
<code>\$shm_save;</code>	Saves the database.
<code>\$shm_close;</code>	Closes the database.

The following sample shows a simple example of how to add signals to an SHM database.



You can insert this code sample into your Verilog HDL file. It is applicable only to Verilog HDL files.

```
initial  
begin  
    $shm_open ("waves.shm");  
    $shm_probe ("AS");  
end
```



For more information on these system tasks, see the Cadence NC-Sim user manual included in the installation.

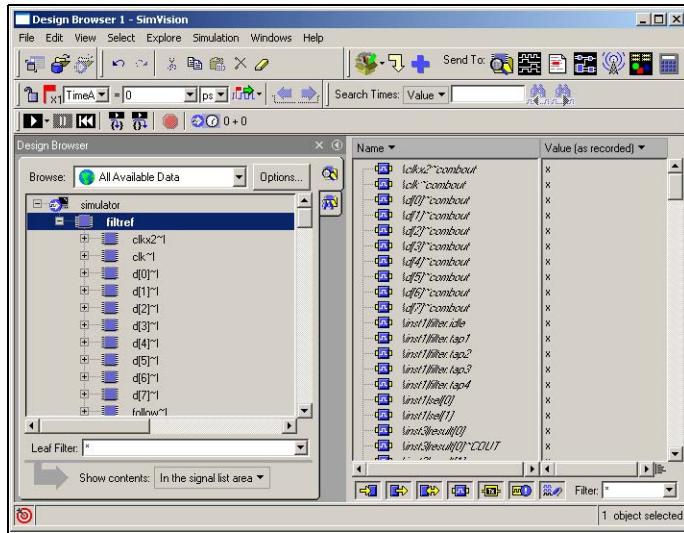
Adding Signals in GUI Mode

To add signals in GUI mode, perform the following steps:

1. Load the design.
 - a. In the nclaunch window, click the + icon to expand the **Snapshots** directory.
 - b. Select the **lib.cell:view** you want to simulate, and choose **NCSim** (right button pop-up menu).
 - c. Click **OK** in the **Simulate** dialog box.

After you load the design, the SimVision Console and SimVision Design Browser windows appear. [Figure 3–5](#) shows the SimVision Design Browser window.

Figure 3–5. SimVision Design Browser



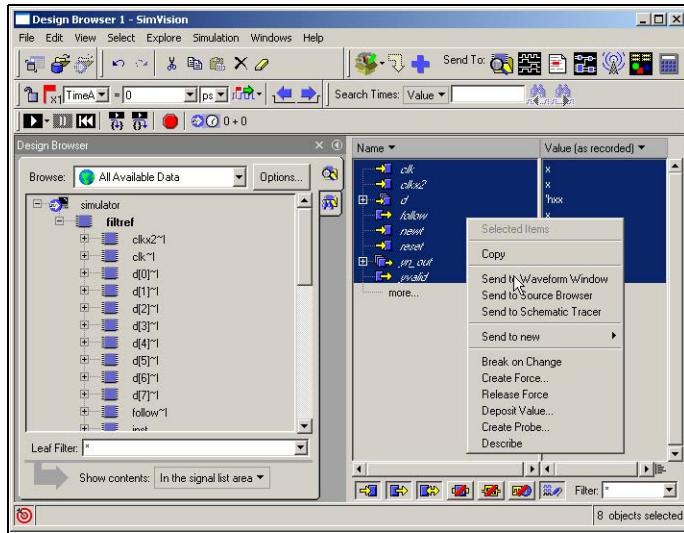
2. In the Design Browser window, select a module in the left side of the window to display the signal names in the module ([Figure 3–5](#)).
 3. To send the selected signals to the Waveform Viewer, perform one of the following steps:

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

Choose **Send to Waveform Window** from the right button pop-up menu, as shown in Figure 3-6.

A waveform window appears that shows all of your signals. You are now ready to simulate your testbench and design.

Figure 3–6. 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 VCD.

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 3–5 shows some of the options you can use with **ncsim**.

Table 3–5. 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, choose **Run** (Simulation menu) to begin the simulation.



You must load the design before simulating. If you have not done so, see step 1 in “[Adding Signals in GUI Mode](#)” on [page 3–17](#) 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.

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. Choose **EDA Tools Settings** (Assignments menu).
2. Choose **Simulation** from the **Category** list or double-click **Simulation** in the EDA Tools Settings section of the **Settings** dialog box.
3. Choose **NC-Verilog** (Verilog) or **NC-VHDL** (VHDL) in the Tools name list.

4. Turn on **Generate netlist for functional simulation only**. If you want to use a testbench or run a script with your netlist, click **More Settings** and enter the required information.



You can also use NativeLink® to launch the simulator automatically after compilation. NativeLink is used if you turn on **Run this tool automatically after compilation**.

5. Click **OK**.
6. Choose **Start Compilation** (Processing menu). If you have previously completed a compilation, choose **Start EDA Netlist Writer** (Processing > Start menu) to generate the functional netlist.



The simulation netlist output is located in the `/<your project directory>/simulation/ncsim` directory.

Set Up Your Environment

Set up your working environment for the Quartus II software and Incisive platform interface. See the instructions in the section “[Set Up Your Environment](#)” on page [3–5](#) for details.

Create Libraries

Create the following libraries for your simulation:

- A work library
- A library for the device family your design targets using the following files in the `<Quartus II installation>/eda/sim_lib` directory:
 - `<device_family>.atoms.v`
 - `<device_family>.atoms.vhd`
 - `<device_family>.components.vhd`

Compile the Project Files & Libraries

Compile the project files and libraries into your work directory using the **ncvlog** or **ncvhdl** programs or the GUI. Include the following files:

- Testbench file
- The Quartus II functional output netlist file (VO or VHO)
- Atom library file for the device family
<device family>_atoms.<v | vhd>
- For VHDL, *<device family>_components.vhd*



See “[Compile Source Code & Testbenches](#)” on page 3–12 for instructions on compiling.

Elaborate Your Design

Elaborate your design using the **ncelab** program as described in “[Elaboration in GUI Mode](#)” on page 3–15.

Add Signals to View

See the section “[Add Signals to View](#)” on page 3–16 for information on adding signals to view.

Simulate Your Design

Simulate your design using the **ncsim** program as described in “[Simulate Your Design](#)” on page 3–19.

Gate-Level Timing Simulation

The following sections provide detailed instructions for performing timing simulation using the Quartus II output files and simulation libraries and Cadence NC tools.

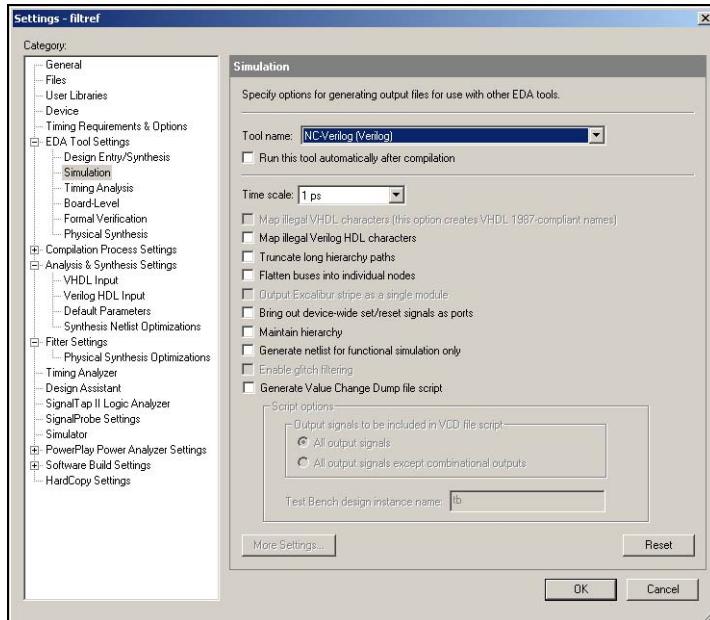
Quartus II Simulation Output Files

When you compile your design, the Quartus II software generates VO or VHO files and an SDO file that are compatible with Cadence NC simulators. To generate these files, perform the following steps in the Quartus II software.

1. Choose **EDA Tool Settings** (Assignments menu).
2. Choose **Simulation** from the **Category** list or double-click **Simulation** in the EDA Tools Settings section of the **Settings** dialog box.

3. In the **Simulation** page of the **Settings** dialog box (Figure 3–7), select NC-Verilog (Verilog HDL output from Quartus II) or NC-VHDL (VHDL output from Quartus II) in the Tool name list.

Figure 3–7. Quartus II EDA Tool Settings



4. Click **OK**.
5. Choose **Start Compilation** (Processing menu).

During compilation, the Quartus II software automatically creates the directory **simulation/ncsim**, which contains the VO, VHO, and SDO files for timing simulation.

Quartus II Timing Simulation Libraries

Altera device simulation library files are provided in the *<Quartus II installation>/eda/sim_lib* directory. The VO or VHO file requires the library for the device your design targets. For example, the Stratix® device family has the following libraries:

- **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. See “[Create Libraries](#)” on page 3–24 for more information.

Set Up Your Environment

Set up your working environment for the Quartus II/NC-Verilog or NC-VHDL software interface. See the section “[Set Up Your Environment](#)” on page 3–5 for details.

Create Libraries

Create the following libraries for your simulation:

- A work library
- The library for the device family your design targets using the following files in the *<Quartus II installation>/eda/sim_lib* directory:
 - <device_family>_atoms.v**
 - <device_family>_atoms.vhd**
 - <device_family>_components.vhd**
- If your design uses the high-speed gigabit transceiver block, you need to create a **stratix_gxb** library and compile the following library files located in the atom *<Quartus II installation>/eda/sim_lib* directory:
 - stratixgx_hssi_atoms.v**
 - stratixgx_hssi_atoms.vhd**
 - stratixgx_hssi_components.vhd**

The **stratix_gxb** library uses the **LPM** and **SGATE** libraries. You can use the following files in the *<Quartus II installation>/eda/sim_lib* directory to create the LPM and SGATE libraries:

220model.v
220model.vhd
220pack.vhd
sgate.v
sgate.vhd
sgate_pack.vhd



See “[Basic Library Setup](#)” on page 3–7 and “[LPM Function & Altera Megafunction Libraries](#)” on page 3–9 for step-by-step instructions on creating libraries.

Compile the Project Files & Libraries

Compile the project files and libraries into your work directory using the **ncvlog** or **ncvhdl** programs or the GUI. Include the following files:

- Testbench file
- The Quartus II output netlist file (VO or VHO)
- Atom library file for the device family
<device family>_atoms.<v | vhd>
- For VHDL, *<device family>_components.vhd*



See “[Compile Source Code & Testbenches](#)” on page 3–12 for instructions on compiling.

Elaborate Your Design

When performing elaboration with the Quartus II-generated Verilog HDL netlist file, the SDO file is read automatically. When you run **ncelab**, it recognizes the embedded system task **\$sdf_annotation** and automatically compiles and annotates the SDO file (runs **ncsdfc** automatically).



See “[Elaborate Your Design](#)” on page 3–14 for step-by-step instructions on elaboration.

For VHDL, the Quartus II-generated VHDL netlist file does not have system task calls to locate your SDF file. Therefore, you must compile the SDO file manually. See “[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 on compiling the SDO file.

Compiling the Standard Delay Output File (VHDL Only) in Command-Line Mode

To annotate the SDO timing data from the command line, perform the following steps:

1. Compile the SDO 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 SDF output file.



If you do not specify an output name, **ncsdfc** uses *<project name>.sdo.X*.

2. Specify the compiled SDO file for the project by adding the following lines to an ASCII SDF command file for the project:

```
COMPILED_SDF_FILE = "<project name>.sdf.X" SCOPE =  
<instance path>
```

The following sample shows an example 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 SDO file, run the following command to elaborate the design:

```
nclab 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 timing data in the GUI, perform the following steps in the nclaunch window:

1. Choose **SDF Compiler** (Tools menu).
2. In the **SDF File** box, type in the name of the SDO file for the project.

3. Click **OK**.

When you finish compiling the SDO file, you can elaborate the design. See “[Elaboration in GUI Mode](#)” on page 3–15 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, do steps 4 through 10.

4. Click **Advanced Options** in the **Elaborate** dialog box.
5. Click **Annotation** in the left side of the dialog box.
6. Turn on the **Use SDF File** option in the right side of the dialog box.
7. Click **Edit**.
8. Browse to the location of the SDF command file name.
9. Browse to the location of the SDO file in the **Compiled SDF File** box and click **OK**.
10. Click **OK** to save and exit the **SDF Command File** dialog box.

Add Signals to View

See the section “[Add Signals to View](#)” on page 3–16 for information on adding signals to view.

Simulate Your Design

Simulate your design using the **ncsim** program as described in “[Simulate Your Design](#)” on page 3–19.

Incorporating PLI Routines

Designers frequently use programming language interface (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 a HEX file for memory, you can convert it for use with NC tools using the Altera-provided **convert_hex2ver** function. However, before you can 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 `<Quartus II installation>/eda/cadence/verilog-xl` directory:

- `convert_hex2ver.c`
- `veriuser.c`
- `convert_hex2ver.obj`

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 that is provided with the Quartus II software.

The `veriuser.c` file is located in the following directory:

`<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. When asked if you want 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 NC-Sim, perform the following steps:

1. Modify the **veriuser.c** file located in the following directory:

<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 by selecting **PLI Wizard** (Utilities menu) in the NC Launch 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, 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. When asked if you want 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:

```
ncelab worklib.my1pmrom -FILE cmd_file.nc ↵
```

- Use the **cmd_file.xl** command file with **verilog-xl** or **ncverilog** to perform your simulations:

```
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, perform the following steps:

1. Run the PLI wizard by typing **pliwiz** at the command prompt or by selecting **PLI Wizard** (Utilities menu) in the NC Launch 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 the **Browse** button to locate the **veriuser.c** file that is provided with the Quartus II software.

The **veriuser.c** file is found in the following location:

<*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, choose 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> ↵
```

Example:

```
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 of these procedures at a command prompt.

For detailed information about specific scripting command options and Tcl API packages, type **quartus_sh --qhelp** at a system command prompt to run the Quartus II Command-Line and Tcl API Help utility.



For more information on Quartus II scripting support, including examples, refer to the *Tcl Scripting* and *Command-Line Scripting* chapters in Volume 2 of the *Quartus II Handbook*.

Generate NC-Sim Simulation Output Files

You can generate VO and SDO simulation output files with Tcl commands or at a command prompt.



For more information about generating VO and SDO simulation output files, refer to “[Quartus II Simulation Output Files](#)” on page 3–22.

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 HDL output from Quartus II)"
```

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 simulator. Specify Vhdl or Verilog HDL 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 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, you 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.

References

- Cadence NC-Verilog Simulator Help
- Cadence NC VHDL Simulator Help
- Cadence NC Launch User Guide

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 Prime Time 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 Prime Time on your Quartus II software designs, and export a netlist, timing constraints, and libraries to the Prime Time 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 Prime Time on your Quartus designs.

This section includes the following chapters:

- [Chapter 4, Simulating Altera IP in Third-Party Simulation Tools](#)
- [Chapter 5, Quartus II Timing Analysis](#)

Revision History

Chapter 5, *Synopsis Prime Time Support* was moved to section III, Volume 3 of the *Quartus II Handbook*.

The table below shows the revision history for [Chapter 4](#) and [5](#).

Chapter(s)	Date / Version	Changes Made
4	Dec. 2004 v1.0	Initial release.
5	Jan. 2005 v2.2	Updated information pertaining to realistic, optimistic, and pessimistic settings
	Dec. 2004 v2.1	<ul style="list-style-type: none">● Chapter 5 was formerly Chapter 4.● Updates to tables and figures.● New functionality for Quartus II software 4.2.
	June 2004 v2.0	<ul style="list-style-type: none">● Updates to tables and figures.● New functionality for Quartus II software 4.1.
	Feb. 2004 v1.0	Initial release.



4. Simulating Altera IP in Third-Party Simulation Tools

qii53014-1.0

Introduction

As both the capacity and complexity of Altera® FPGA devices continues to increase, 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 (AMPPSM) offer a broad portfolio of IP megafunctions optimized for Altera FPGAs. Through parameterization, these reusable blocks of intellectual property (IP) can be customized to meet your design requirements.

Even when the IP source code is encrypted or otherwise restricted, Altera's unique IP Toolbench allows you to efficiently simulate IP designs. Using IP Toolbench, you can custom configure IP designs and then generate a VHDL or Verilog functional simulation model for use in your choice of simulation tools.

This handbook chapter provides you with an overview of the process for generating an IP functional simulation model in IP Toolbench and simulating this model in an Altera-supported, third-party simulation tool.

Generating an IP Functional Simulation Model with IP Toolbench

IP megafunctions deployed through IP Toolbench allow you to quickly and easily view documentation, specify parameters, set up third-party tools, and generate all of the necessary files for integrating the parameterized IP megafunction in your design. IP Toolbench can be launched from within the Quartus® II software via the MegaWizard® Plug-In Manager (Tools menu) (Figure 4-1).

Figure 4–1. IP Toolbench GUI

The IP Toolbench graphical user interface (GUI) may appear differently depending on the IP megafunction you are using, or whether you select an IP megafunction from the MegaWizard Plug-In Manager or SOPC Builder. IP Toolbench GUIs from SOPC Builder do not include buttons for setting up simulation or generation, because those functions are performed by SOPC Builder when the entire system is generated. This section describes the procedure for using an IP Toolbench-supported IP megafunction chosen in the MegaWizard Plug-In Manager.

After parameterizing your megafunction, IP Toolbench, in conjunction with the Quartus II software, generates a Verilog Output File (.vo) or VHDL Output File (.vho) IP functional simulation model. These high-level output files differ from the low-level VO and VHO models generated by the Quartus II software for post-synthesis or post place-and-route simulation.

Low-Level VO or VHO Simulation Models

The low-level VO or VHO simulation model files (generated after synthesis or place-and-route) are mapped to atoms. An atom is a parameterized, device-family-dependent representation of a WYSIWYG primitive that corresponds to a device feature such as a logic element (LE), an I/O element (IOE), or memory. Altera-supported simulators include libraries of atoms for each device family. The VO or VHO models that are generated for post-synthesis or post place-and-route simulation are placed in the simulation folder under the Quartus II project folder.

High-Level VO or VHO Simulation Models

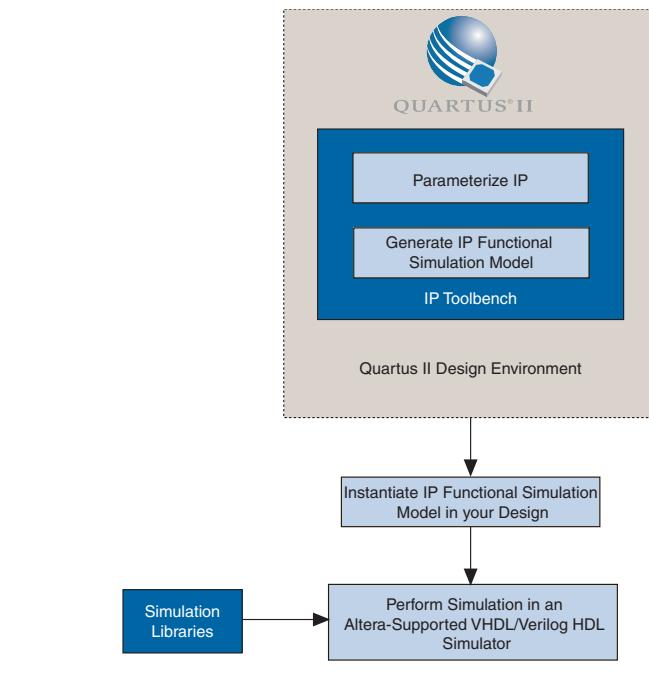
The IP functional simulation models generated through IP Toolbench are mapped to higher-level primitives such as adders, multipliers, and multiplexers. This higher level of abstraction results in much faster simulation times. Depending on the megafunction, the IP functional simulation model is up to 100 times faster than a Quartus II Post-synthesis or post place-and-route simulation. The VO or VHO models that are generated for IP functional simulation purposes are output to the directory that you specify in IP Toolbench, typically the same folder as your Quartus II project. These simulation models can be used in your Altera-supported simulator along with other models and HDL source code.

You can use the IP functional simulation model only for simulation purposes, and not for synthesis or any other purpose. Attempting to synthesize the IP functional simulation model in Quartus II software, results in a non-functional design.



Generating an IP functional simulation model for Altera MegaCore functions does not require a license. However, generating an IP functional simulation model for AMPP megafunctions may require a license. For more information, contact the IP megafunction vendor.

Figure 4–2 outlines the process of generating an IP functional simulation model and simulating your design in an Altera-supported VHDL or Verilog HDL simulator.

Figure 4–2. IP Functional Simulation Model Design Flow

Launch IP Toolbench

To launch IP Toolbench in the Quartus II software, follow these steps:

1. Start the MegaWizard Plug-In Manager by choosing **MegaWizardPlug-In Manager** (Tools menu). The **MegaWizard Plug-In Manager** dialog box is displayed.



Refer to the Quartus II Help for more information on how to use the MegaWizard Plug-In Manager.

2. Choose **Create a new custom megafunction variation**. Click **Next**.
3. Select the megafunction you would like to create.
4. Choose the output file type for your design, and specify the name and location of the file to be generated.
5. Click **Next** to launch IP Toolbench for the megafunction you have selected.

Step 1: Parameterize

To create a custom variation of the megafunction, follow these steps:

1. Click the **Step 1: Parameterize** button in IP Toolbench ([Figure 4–3](#)).

Figure 4–3. Step 1: Parameterize



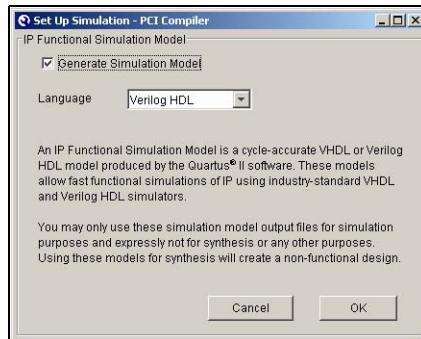
[Figure 4–3](#) shows the IP Toolbench for the PCI Compiler, v3.0.0. The number of buttons in the IP Toolbench—and their names—may vary per megafunction.

Step 2: Set Up Simulation

After the megafunction variation has been parameterized, click the **Step 2: Set Up Simulation** button in IP Toolbench ([Figure 4–4](#)).

Figure 4–4. Step 2: Set Up Simulation

-
2. Turn on **Generate Simulation Model** (Figure 4–5).

Figure 4–5. Set Up Simulation Dialog Box

-
3. Choose the language in the **Language** list.
 4. Click **Ok**.

Step 3: Generate

To generate your megafunction, follow these steps:

1. Click the **Step 3: Generate** button to generate the custom megafunction variation and the IP functional simulation model (Figure 4–6).

Figure 4–6. Generate

When you click **Generate**, IP Toolbench creates several new design files in the directory specified in the MegaWizard Plug-In Manager ([Table 4–1](#)). The files vary based on the HDL language selected for your custom megafunction and IP functional simulation model. IP Toolbench may generate additional files for other purposes, depending on the IP megafunction you are using.

Table 4–1. IP Toolbench-Generated Files

Extension	Description
.vhd, or .v	A megafunction variation file, which defines a VHDL or Verilog HDL top-level description of the custom megafunction. Instantiate the entity defined by this file inside of your design. Include this file when compiling your design in the Quartus II software.
.cmp	A VHDL component declaration file for the megafunction variation. Add the contents of this file to any VHDL architecture that instantiates the megafunction.
.inc	An AHDL include declaration file for the megafunction variation. Include this file with any AHDL architecture that instantiates the megafunction.
_bb.v	A Verilog HDL black-box file for the megafunction variation. Use this file when using a third-party EDA tool to synthesize your design.
.bsf	A Quartus II symbol file for the megafunction variation. You can use this file in the Quartus II block diagram editor.
.html	A megafunction report file.
.vo or .vho	A VHDL or Verilog HDL IP functional simulation model.
_inst.vhd or _inst.v	A VHDL or Verilog HDL sample instantiation file.

Step 4: Instantiate the IP Functional Simulation Model in Your Design

When using the IP functional simulation model to simulate your design, you do not have to make any changes to your original design files. The only difference is which files you add to the simulation and synthesis projects. To perform the IP functional simulation, add the IP functional simulation model VHO or VO file to your simulation project.

To synthesize your design using the Quartus II software, add the IP Toolbench-generated verilog or VHDL custom variation file to your Quartus II project.

To synthesize your design using a third-party EDA tool, add the IP Toolbench-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.

Step 5: Perform Simulation

The IP functional simulation model that is generated by IP Toolbench instantiates high-level primitives such as adders, multipliers, and multiplexers; LPM functions, and Altera megafunctions.

To properly compile, load, and simulate the IP functional simulation model generated by the Quartus II software, you must first compile the **s gate**, **altera_mf**, and **220model** libraries in your simulation tool.

- The **s gate** library includes the definition of the high-level primitives
- The **altera_mf** includes the definition of Altera megafunctions
- The **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 already compiled and mapped; thus, you do not need to compile them.

Figure 4–7 illustrates how libraries are used in IP functional simulation.

Figure 4–7. IP Functional Simulation Library Usage

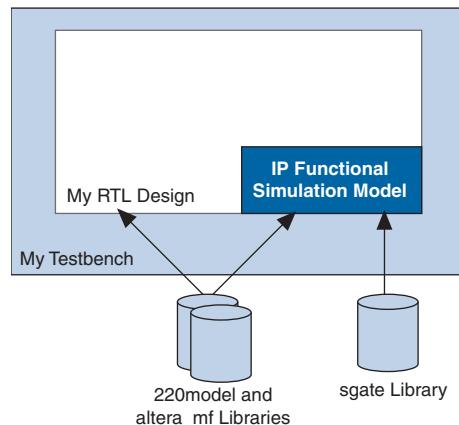


Table 4–2 lists the simulation library files, where *<path>* is the directory where the Quartus II software is installed.

Table 4–2. Simulation Library Files

Location	HDL Simulation	Description
<i><path>/eda/sim_lib/sgate.v</i>	Verilog HDL	Libraries that contain simulation models for IP functional models
<i><path>/eda/sim_lib/sgate.vhd</i>	VHDL	
<i><path>/eda/sim_lib/sgate_pack.vhd</i>	VHDL	Libraries that contain VHDL component declarations for the sgate.vhd library
<i><path>/eda/sim_lib/220model.v</i>	Verilog HDL	Libraries that contain simulation models for the Altera library of parameterized modules (LPM) version 2.2.0
<i><path>/eda/sim_lib/220model.vhd</i>	VHDL	
<i><path>/eda/sim_lib/220pack.vhd</i>	VHDL	Libraries that contain VHDL component declarations for the 220model.vhd library
<i><path>/eda/sim_lib/altera_mf.v</i>	Verilog HDL	Libraries that contain simulation models for Altera-specific megafunctions
<i><path>/eda/sim_lib/altera_mf.vhd</i>	VHDL	
<i><path>/eda/sim_lib/altera_mf_components.vhd</i>	VHDL	Libraries that contains VHDL component declarations for the altera_mf.vhd library

Design Language Examples

This section provides the following design language examples:

- ModelSim Verilog
- ModelSim VHDL
- NC-VHDL
- VCS

Verilog Example: Simulating the IP Functional Simulation Model in the ModelSim Software

The following example shows the process of simulating a Verilog-based, megafunction. The example assumes that the megafunction variation and the IP functional simulation model have been generated.

1. Create a ModelSim project by performing the following steps:
 - a. In the ModelSim software, select **New > Project** (File menu).
 - b. In the **Create Project** dialog box, 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 IP functional simulation model generated by IP Toolbench. (If you are using the ModelSim-Altera software, skip to step 5.)
 - The **sgate.v**, **220model.v**, and **altera_mf.v** library files.

2. To create the required simulation libraries, type the following commands at the ModelSim prompt:

```
vlib sgate <
```

```
vlib lpm <
```

```
vlib altera_mf <
```

3. To map to the required simulation libraries, type the following commands at the ModelSim prompt:

```
vmap sgate sgate <
```

```
vmap lpm lpm
vmap altera_mf altera_mf
```

4. To compile the HDL into libraries, type 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. To compile the IP functional simulation model, type the following command at the ModelSim prompt:

```
vlog -work work <my_IPtoolbench_output_netlist>.vo
```

6. To compile your RTL, type the following command at the ModelSim prompt:

```
vlog -work work <my_design>.v
```

7. To compile the testbench, type the following command at the ModelSim prompt:

```
vlog -work work <my_testbench>.v
```

8. To load the testbench, type 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 IP Functional Simulation Model in the ModelSim Software

The following example shows the process of performing a functional simulation of a VHDL-based, megafunction IP functional simulation model. The example assumes that the megafunction variation and the IP functional simulation model have been generated.

1. Create a ModelSim project by performing the following steps:
 - a. In the ModelSim software, select **New > Project** (File menu).

- b. In the **Create Project** dialog box, 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 relevant files to your simulation project:
 - Add your design files
 - Add the IP functional simulation model generated by IP Toolbench. (If you are using the ModelSim-Altera software, skip to step 5.)
 - Add the **sgate.vhd**, **sgate.vhd**, **sgate_pack.vhd**, **220model.vhd**, **220pack.vhd**, **altera_mf.vhd**, and **altera_mf_components.vhd** library files.
2. To create the required simulation libraries, type the following commands at the ModelSim prompt:

```
vlib sgate <br/>vlib lpm <br/>vlib altera_mf <br/>
```
 3. To map to the required simulation libraries, type the following commands at the ModelSim prompt:

```
vmap sgate sgate <br/>vmap lpm lpm <br/>vmap altera_mf altera_mf <br/>
```

4. To compile the HDL into libraries, type the following commands at the ModelSim prompt:

```
vcom -work altera_mf -93 -explicit  
altera_mf_components.vhd <br/>  
vcom -work altera_mf -93 -explicit altera_mf.vhd <br/>  
vcom -work lpm -93 -explicit 220pack.vhd <br/>  
vcom -work lpm -93 -explicit 220model.vhd <br/>
```

```
vcom -work sgate -93 -explicit sgate_pack.vhd ↵  
vcom -work sgate -93 -explicit sgate.vhd ↵
```

5. To compile the IP functional simulation model, type the following command at the ModelSim prompt:

```
vcom -work work -93 -explicit <output netlist>.vho ↵
```

6. To compile the RTL, type the following command at the ModelSim prompt:

```
vcom -work work -93 -explicit <RTL>.vhd ↵
```

7. To compile the testbench, type the following command at the ModelSim prompt:

```
vcom -work work -93 -explicit <my testbench>.vhd ↵
```

8. To load the testbench, type the following command at the ModelSim prompt:

```
vsim work.my_testbench ↵
```

NC-VHDL Example: Simulating the IP Functional Simulation Model in the NC-VHDL Software

The following example shows the process of performing a functional simulation of a NC-VHDL-based, megafunction IP functional-simulation model. The example assumes that the megafunction variation and the IP functional simulation model have been generated.

1. To create a **cds.lib** file, type the following entries:

```
DEFINE worklib ./worklib  
DEFINE sgate ./sgate  
DEFINE altera_mf ./altera_mf  
DEFINE lpm ./lpm
```

2. To compile library files into appropriate libraries, type 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. To compile source code and testbench files, type 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. To elaborate the design, type the following command at the command prompt:

```
ncelab worklib.<my_testbench>:entity ↵
```

Verilog HDL Example: Simulating Your IP Functional Simulation Model in VCS

The following example illustrates the process of performing a functional simulation of a design that contains a Verilog HDL-based, megafunction IP functional simulation model. This example assumes that the megafunction variation and the IP functional simulation model have been generated.

Single Step Process

For the single-step process, type the following at a command prompt:

```
vcs <testbench>.v <RTL>.v <output netlist>.v -v 220model.v
altera_mf.v sgate.v -R ↵
```

Two-Step Process (Compilation & Simulation)

For compilation and simulation, perform the following steps:

1. To compile your design files, type the following at a command prompt:

```
vcs <testbench>.v <RTL>.v <output netlist>.v -v 220model.v  
altera_mf.v sgate.v -o simulation_out ↵
```

2. To load your simulation, type the following at a command prompt:

```
source simulation_out ↵
```



For more information on simulating a design in VCS, refer to the *Synopsys VCS Support* chapter in Volume 3 of the *Quartus II Handbook*.

Conclusion

Using the Quartus II software and IP Toolbench, you can generate IP functional simulation models that enable you to efficiently simulate your design. The high level of abstraction of the IP functional simulation model—relative to post-synthesis or post place-and-route netlists—results in fast behavioral simulations of megafunctions. Using an IP functional simulation model is also transparent, requiring only adding different files to synthesis and simulation projects. These features enhance and simplify design verification.

qii53004-2.2

Introduction

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. Timing analysis measures the delay of every design path and reports the performance of the design in terms of the maximum clock speed. However, it does not check design functionality and should be used together with simulation to verify the overall design operation.

The Quartus® II software provides the features necessary to perform advanced timing analysis for today's system-on-a-programmable-chip (SOPC) designs. During compilation the Quartus II software automatically performs timing analysis so that you don't have to launch a separate timing analysis tool after each successful compilation. The Quartus II Timing Analyzer reports timing analysis results in the Compilation Report, giving you immediate access to this data.

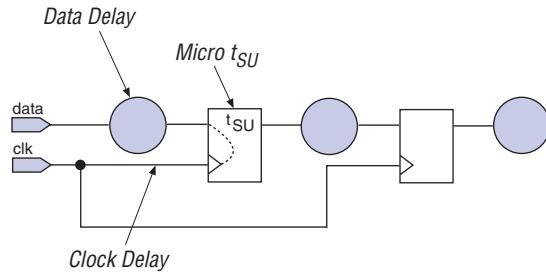
This chapter explains the basic principles of static timing analysis, and the advanced features supported by the Quartus II Timing Analyzer using TCL scripts and the Quartus II graphical user interface (GUI).

Timing Analysis Basics

A comprehensive timing analysis involves observing the setup times, hold times, clock-to-output delays, maximum clock frequencies, and slack times for the design. With this information you can validate circuit performance and detect possible timing violations. Undetected timing violations can result in incorrect circuit operation. This section describes the basic timing analysis measurements made by the Quartus II Timing Analyzer.

Clock Setup Time (t_{SU})

Data that feeds a register's data or enable inputs must arrive at the input pin before the register's clock signal is asserted at the clock pin. Clock setup time is the minimum length of time that data must be stable before the active clock edge. [Figure 5-1](#) shows a diagram of clock setup time.

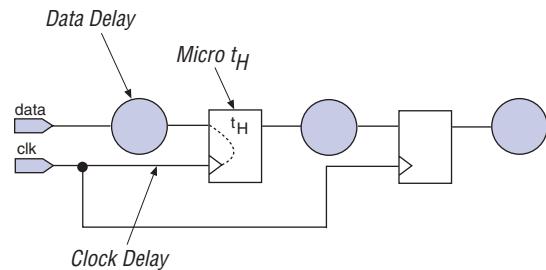
Figure 5–1. Clock Setup Time (t_{SU})

Micro t_{SU} is the internal setup time of the register (i.e., it is a characteristic of the register and is unaffected by the signals feeding the register). The following equation calculates the t_{SU} of the circuit shown in Figure 5–1.

$$t_{SU} = \text{Longest Data Delay} - \text{Shortest Clock Delay} + \text{Micro } t_{SU}$$

Clock Hold Time (t_H)

Data that feeds a register via its data or enable inputs must be held at an input pin after the register's clock signal is asserted at the clock pin. Clock hold time is the minimum length of time that this data must be stable after the active clock edge. Figure 5–2 shows a diagram of clock hold time.

Figure 5–2. Clock Hold Time (t_H)

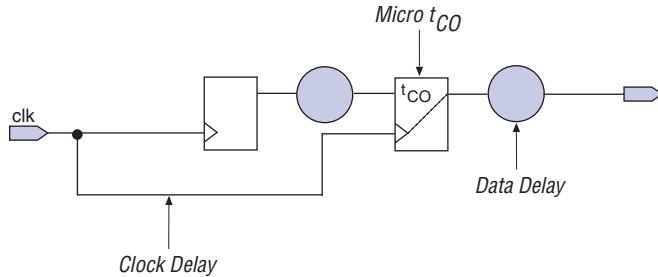
Micro t_H is the internal hold time of the register. The following equation calculates the t_H of the circuit shown in Figure 5–2.

$$t_H = \text{Longest Clock Delay} - \text{Shortest Data Delay} + \text{Micro } t_H$$

Clock-to-Output Delay (t_{CO})

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 5–3 shows a diagram of clock-to-output delay.

Figure 5–3. Clock-to-Output Delay (t_{CO})



The following equation calculates the t_{CO} of the circuit shown in Figure 5–3.

$$t_{CO} = \text{Longest Clock Delay} + \text{Longest Data Delay} + \text{Micro } t_{CO}$$

Pin-to-Pin Delay (t_{PD})

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.



In the Quartus II software, you can make t_{PD} assignments between an input pin and an output pin, an output pin and a register, a register and a register, and a register and an output pin.

Maximum Clock Frequency (f_{MAX})

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 timing analysis on both single and multiple clock designs.



Apply clock settings to all clock nodes in a design to ensure performance requirements are met. Refer to “[Setting up the Timing Analyzer](#)” on page 5–6 for more information.

Slack

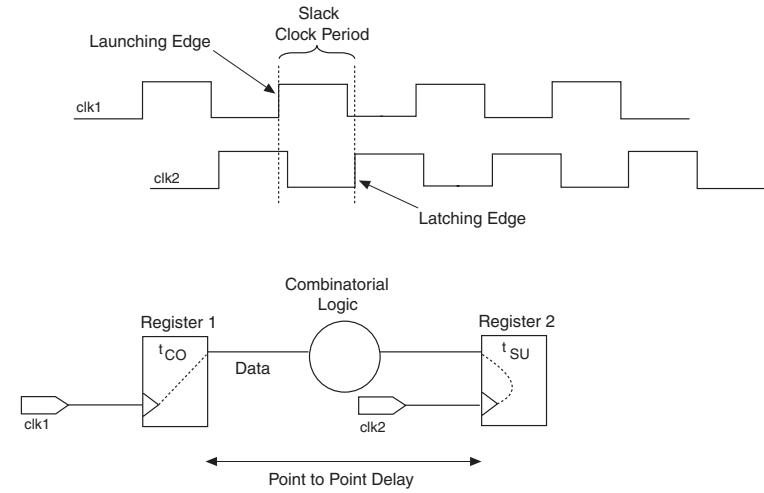
Slack is the margin by which a timing requirement (e.g., f_{MAX}) is met or not met. Positive slack indicates the margin by which a requirement is met. Negative slack indicates the margin by which a requirement was not met. The Quartus II software determines slack with the following equations.

$$\text{Slack} = \text{Longest Point to Point Requirement} - \text{Longest Point to Point Delay}$$

$$\text{Point to Point Requirement} = \text{Setup Relationship} + \text{Clock Skew} - t_{CO} - t_{SU}$$

[Figure 5–4](#) shows a slack calculation diagram.

Figure 5–4. Slack Calculation Diagram

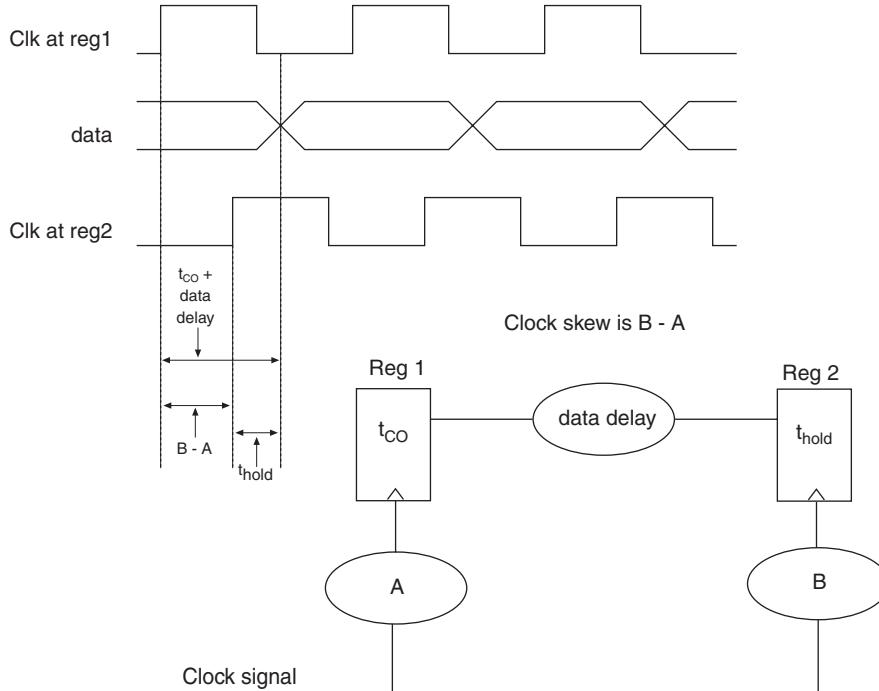


Hold Time Slack

Hold time slack is the margin by which the minimum hold time requirement is met or not met for a register-to-register path ([Figure 5–5](#)). Data is required to remain stable after the rising edge of a destination register’s clock for at least the time equal to the micro hold time of the

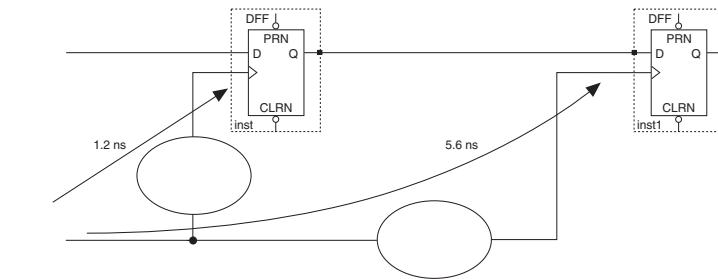
destination register. The primary cause of a hold time violation is excessive clock skew (B - A). As long as the data delay is greater than clock skew (B - A), no hold time violation occurs.

Figure 5–5. Hold Time Slack



Clock Skew

Clock skew is the difference in arrival time of a clock signal at two different registers (Figure 5–6). Clock skew occurs when two clock signal paths have different lengths. Clock skew is common in designs that contain clock signals that are not routed globally. The Quartus II Timing Analyzer reports clock skew for all clocks within the design.

Figure 5–6. Clock Skew

Setting up the Timing Analyzer

You can make certain timing assignments globally for a project, and you can make timing assignments to individual entities in a project. If a project has global and individual timing assignments, the individual timing assignments take precedence over the global timing assignments.

Setting Global Timing Assignments

You can define named clock settings and assign them to a specific node by selecting **Settings for individual clock signals** and clicking **Clocks** on the **Timing Requirements & Options** page of the **Settings** dialog box (Assignments menu) as shown in Figure 5–8 and 5–8.

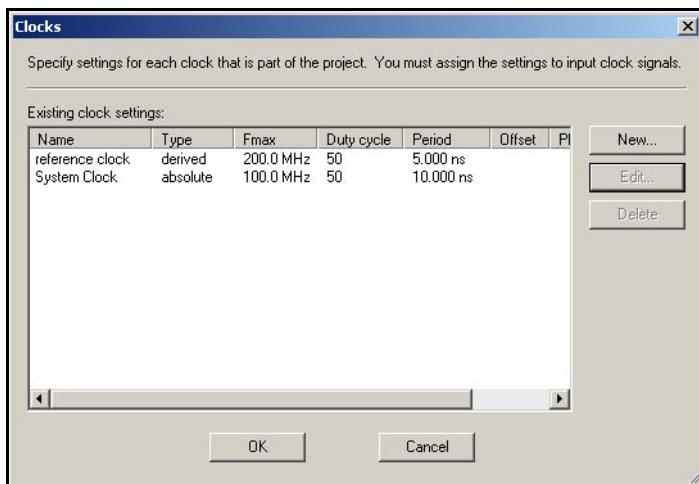
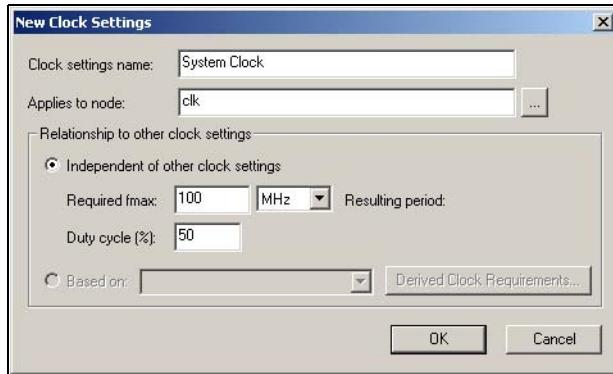
Figure 5–7. Clocks Dialog Box

Figure 5–8. New Clock Settings Dialog Box



You can set global t_{SU} , t_{CO} , and t_{PD} requirements, as well as minimum t_H , t_{CO} , and t_{PD} requirements. You can set a global f_{MAX} requirement, or assign timing requirements and relationships for individual clocks.



For more information about path-cutting options in the **Timing Requirements & Options** page, see “[False Paths](#)” on page 5–29.

Specifying Individual Clock Requirements

Apply clock requirements to each clock in your design. You can define clocks as absolute clocks (independent of other clocks) or derived clocks (dependent on other clocks). To define an absolute clock, you must specify the required f_{MAX} and the duty cycle. A derived clock is based on a previously defined clock. For a derived clock, you can specify the phase shift, offset, and multiplication and division factors relative to the absolute clock. You must define clock requirements and relationships with the **Timing Wizard** or by clicking **Clocks** in the **Timing Requirements & Options** page of the **Settings** dialog box (Assignments menu). Define all clock requirements and relationships in your design to ensure accurate timing analysis results.

Clocks can also be specified using the `create_base_clock` and `create_relative_clock` Tcl commands as shown in [Table 5–1](#).

Clock Type	Command Usage	Example
Absolute	<code>create_base_clock -fmax <fmax> [-duty_cycle <duty cycle>] \ [-target <name>] [-no_target] \ [-entity <entity>] [-disable] \ <clock_name></code>	<code>create_base_clock -fmax 100mhz -duty_cycle 50 clk50 ↵</code>
Relative	<code>create_relative_clock -base_clock <base clock> [-duty_cycle <duty cycle>] [-multiply <number>] [-divide <number>] [-offset <offset>] [-invert] [-target <name>] [-no_target] [-entity <entity>] [-disable] <clock_name></code>	To create a clock <code>Clk2_3</code> created based on a predefined clock <code>clk10</code> : <code>create_relative_clock -base_clock -multiply 2 -divide 3 clk10 clk2_3 ↵</code>

Setting Other Individual Timing Assignments

You can use the Assignment Editor to make other individual timing assignments to pins and nodes in your design.



For detailed information about how to use the Assignment Editor, see the *Assignment Editor* chapter in Volume 2 of the *Quartus II Handbook*. For more detailed information about individual timing assignments, or for information about timing assignments not listed below, see Quartus II Help.

Clock Settings Assignments

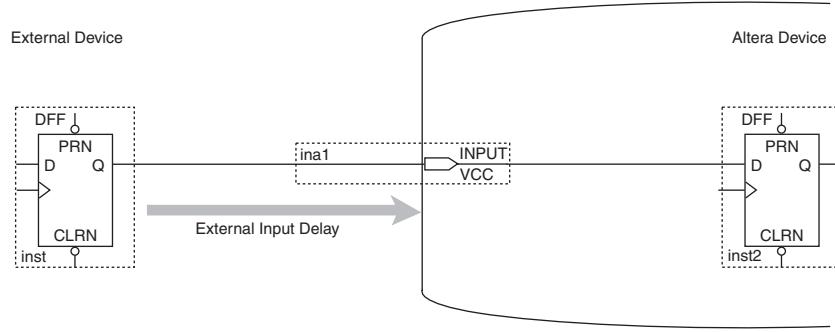
Make **Clock Settings** assignments to assign previously-created clock settings to a pin or node in the design. Named groups of clock settings can be defined in the **Timing Wizard** (Assignments menu) or by selecting **Settings for individual clock signals** and clicking **Clocks** on the **Timing Requirements & Options** page of the **Settings** dialog box (Assignments menu).

Input Maximum Delay & Input Minimum Delay Assignments

Make **Input Maximum Delay** assignments to specify the maximum allowable delay of a signal from a register outside the device to a specified input or bidirectional pin. The value of this assignment usually represents

the t_{CO} of the external register feeding the input pin of the Altera device, plus the actual board delay. Conversely, you can set the minimum allowable delay with the **Input Minimum Delay** assignment. Figure 5–9 shows a block diagram of an external input delay.

Figure 5–9. External Input Delay



In Tcl, after using the `timegroup` command to define the group `input_pins`, an **Input Maximum Delay** assignment of 2 ns can be made to each pin in the group using the `-max` option of `set_input_delay`.

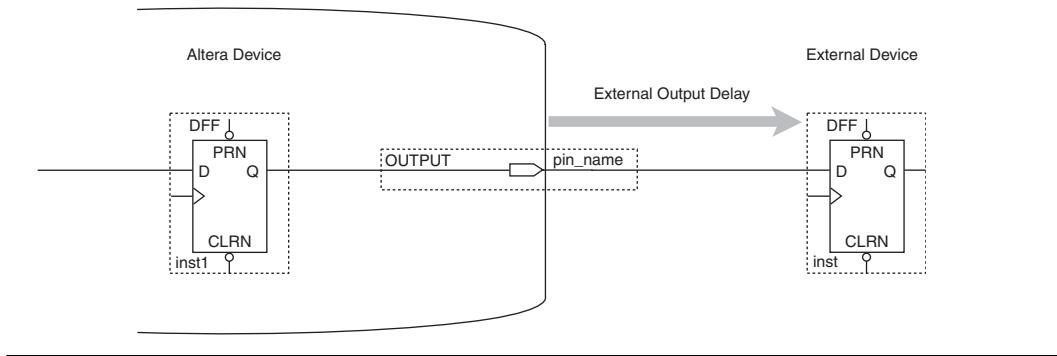
```
timegroup "input_pins" -add_member "i*" -add_exception "ibus*"  
set_input_delay -clk_ref clk -to "input_pins" -max 2ns
```

The assignments created or modified while a project is opened by a Tcl script are not saved in its Quartus II Settings File (.qsf) unless the `export_assignments` command is explicitly executed, or the project is closed using the `close_project` command.

Output Maximum Delay & Output Minimum Delay Assignments

Make **Output Maximum Delay** assignments to specify the maximum allowable delay of a signal from the specified output pin to a register outside the device. The value of this assignment usually represents the t_{SU} of the external register fed by the output pin of the Altera device, plus the actual board delay. Conversely, you can set the minimum allowable delay with the **Output Minimum Delay** assignment. [Figure 5–10](#) shows a block diagram of an external output delay.

Figure 5–10. Output Delay



To make **Output Maximum Delay** and **Output Minimum Delay** assignments in Tcl, use the `set_output_delay` command:

```
set_output_delay [-h | -help] [-long_help]
[-clk_ref <clock>] -to <output_pin> [-min] [-max]
[-clock_fall] [-remove] [-disable]
[-comment <comment>] [<value>]
```



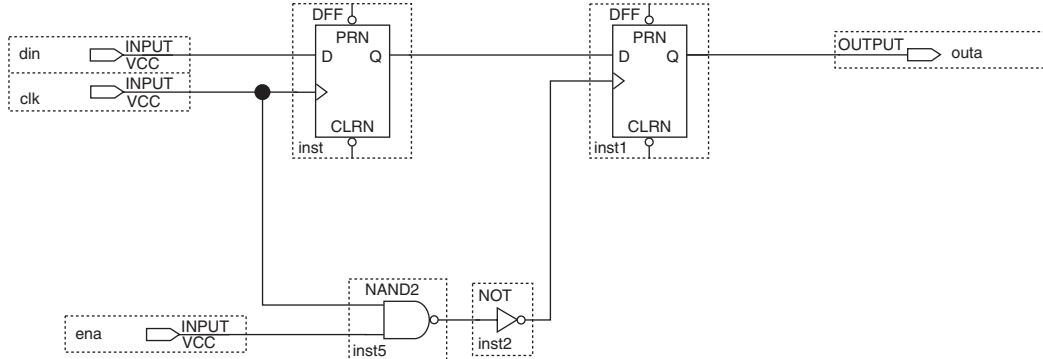
Do not combine individual or global t_{CO} , or t_{SU} assignments with input or output delay assignments.

Inverted Clock Assignments

The Quartus II Timing Analyzer automatically detects registers with inverted clocks and uses the inversion in the timing analysis report. This functionality applies to both clocks that use globals and clocks that do not use globals. However, the Timing Analyzer can fail to automatically detect inverted clocks when the inversion is part of a complex logic structure.

In the complex logic structure shown in Figure 5–11, when the enable is active, the clock is inverted. Under these circumstances, you should make an inverted clock assignment to the register, inst1, to ensure that the Timing Analyzer recognizes the inverted clock.

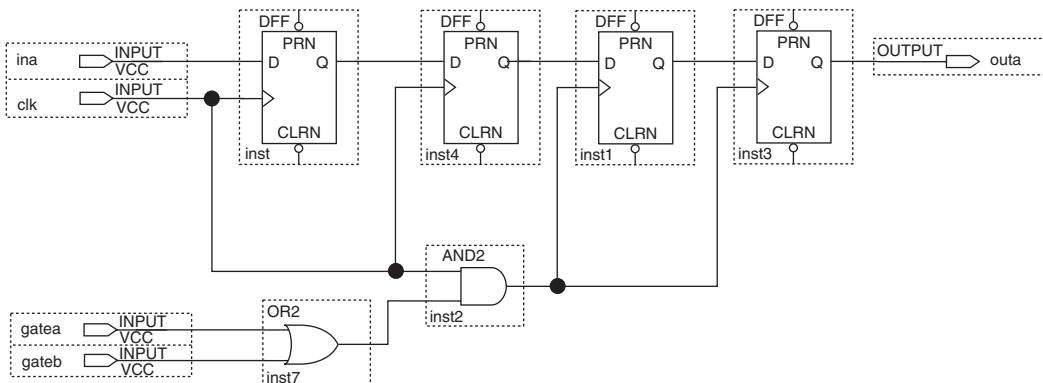
Figure 5–11. Complex Inverted Clock Logic Structure



Not a Clock Assignments

The Timing Analyzer automatically identifies any pin that feeds through to the clock input of a register as a clock. An example is shown in Figure 5–12.

Figure 5–12. Pins Misidentified As Clocks



In [Figure 5-12](#), the Timing Analyzer identifies three clock pins for the design: `clock`, `gatea` and `gateb`. The pins `gatea` and `gateb` are identified as clock pins because they feed through an OR gate and an AND gate to the clock inputs of registers `inst1` and `inst3`. If you do not want to view these pins as clocks, you can remove them from timing analysis with the **Not a Clock** assignment. For example, the following Tcl command explicitly removes the clock signal `clk` from timing analysis:

```
set_instance_assignment -name NOT_A_CLOCK -to clk
```

Maximum Clock Arrival Skew

Allows you to specify the maximum allowable clock skew between a set of registers. Any register identified as a gated or ripple clock (that is, undefined by a clock setting), is ignored in this calculation. However, the calculation can proceed through the ripple clock. You can use the Assignment Editor to assign this requirement from a clock signal to a set of registers defined by a wildcard or time group. After assigning this requirement, the Timing Analyzer reports the results of clock skew analysis in the Maximum Clock Arrival Skew report.

The following Tcl command can be used to apply the Maximum Clock Arrival Skew assignment:

```
set_instance_assignment -name max_clock_arrival_skew  
2ns -from clock -to state_m:inst3*
```

Maximum Data Arrival Skew

Allows you to specify the maximum allowable data skew between a set of registers or pins with respect to a clock signal. The data arrival delay represents the tCO from the clock to the given register and/or pin.

You can use the Assignment Editor to assign this requirement from a clock signal to a set of registers or pins defined by a wildcard or time group. After assigning this requirement, the Timing Analyzer reports the results of data skew analysis in the Maximum Data Arrival Skew report. The following guidelines apply to calculation of this constraint:

- Any input delay on pins feeding input registers is added to the arrival time. However, output delays are not added to output pin arrival times.
- Any register identified as a gated or ripple clock (that is, undefined by a clock setting), is ignored in this calculation. However, the calculation can proceed through the ripple clock.
- This calculation can account for external clocks.

The following Tcl command can be used to apply the Maximum Clock Arrival Skew assignment:

```
set_instance_assignment -name max_data_arrival_skew  
2ns -from clock -to state_m:inst*
```

tco Requirement Assignments

Individual **tco Requirement** assignments override global assignments specifying t_{CO} . You can make a **tco Requirement** assignment to a pin, an output register, or from an output register to a pin.

tsu Requirement Assignments

Individual **tsu Requirement** assignments override global assignments specifying t_{SU} . You can make a **tsu Requirement** assignment to an input pin, an input register, from an input pin to an input register, and from an clock pin to an input register.

-  Do not combine individual or global t_{CO} , or t_{SU} assignments with input or output delay assigments.

th Requirement Assignments

Individual **th Requirement** assignments override global assignments specifying t_H . You can make a **th Requirement** assignment to either a pin, an input register, from a pin to an input register, or from a clock pin to an input register.

tpd Requirement Assignments

Individual **tpd Requirement** assignments override global assignments specifying t_{PD} . You can make a **tpd Requirement** assignment to an input pin, from an input pin to an output pin, from an input pin to a register, from a register to a register, and from a register to an output pin.

-  Global t_{CO} , t_H , t_{PD} , and t_{SU} assignments are made using the **Timing Wizard** (Assignments menu) or on the **Timing Requirements & Options** page of the **Settings** dialog box (Assignments menu).

Timing Analysis Reporting in the Quartus II Software

The Quartus II timing analysis report is displayed as a section of the Compilation Report. The Timing Analysis Report includes f_{MAX} and slack measurements for all clock pins. The report 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.



If there are no timing assignments for the design, the Timing Analyzer does not generate slack reports for the clock pins.

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.

The Quartus II software only reports slack measurements for pins with individual or global t_{SU} , t_H or t_{CO} assignments.

Advanced Timing Analysis



The Quartus II software performs timing analysis of designs that contain several clocks. This section also describes Multicycle assignments and how the relevant timing is measured by the Quartus II Timing Analyzer.

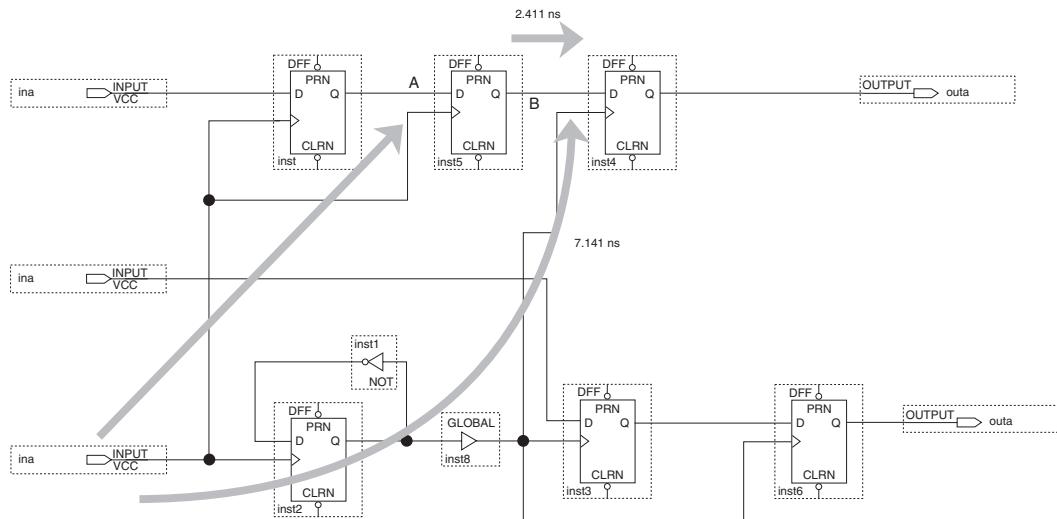
For detailed instructions on how to use these or any of the Quartus II Timing Analyzer features, see the Quartus II Help.

Clock Skew

This section describes some common cases in which clock skew may result in incorrect circuit operation.

Derived Clocks

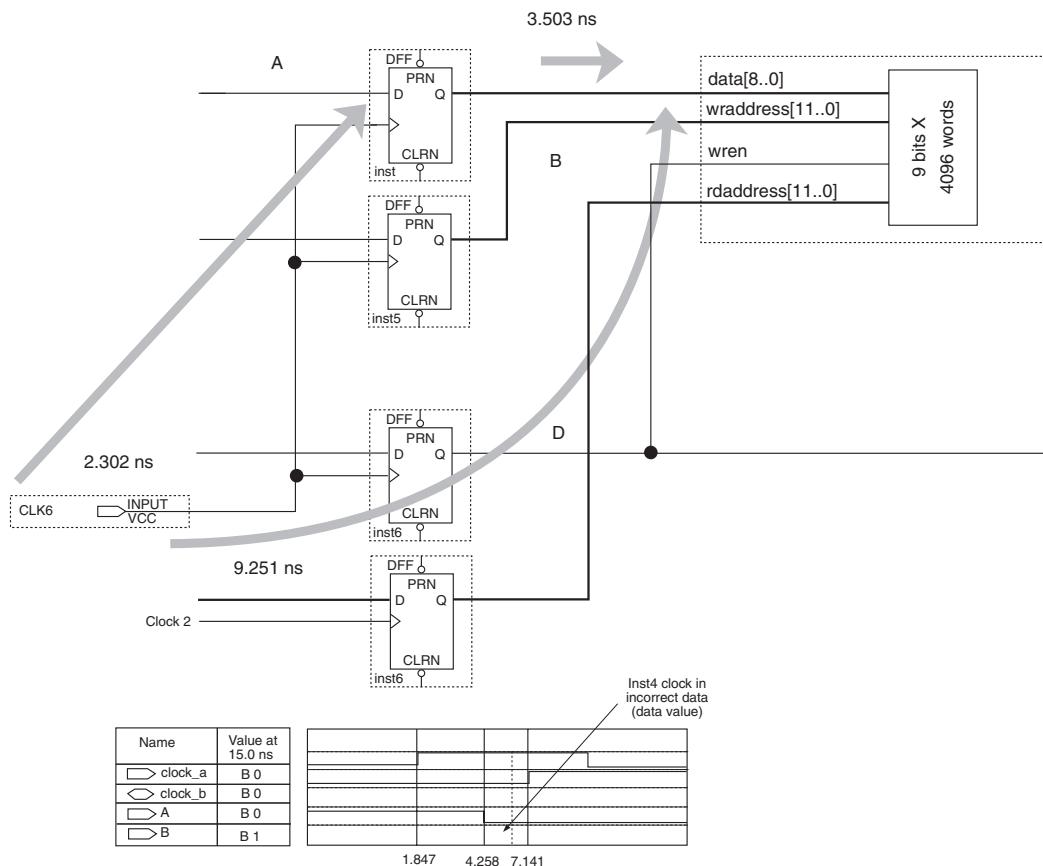
Clock skew error reporting may occur in designs containing derived clocks and very short register-to-register data paths. In [Figure 5–13](#), the longest clock path is 7.141 ns from `clock_a` to destination register `inst4`. The shortest clock path is 1.847 ns from `ina` to the source register `inst5`. This creates a clock skew of 5.294 ns.

Figure 5–13. Derived Clocks Example

The shortest register-to-register data path between the source and destination register is 2.411 ns. Thus, the clock skew is longer than the data path ($5.294 \text{ ns} > 2.411 \text{ ns}$). This results in incorrect circuit functionality. To remove the clock skew error, path B must be lengthened so that it is longer than the clock skew. This is achieved by adding cells to the path or through the placement of the source and destination registers.

Asynchronous Memory

With asynchronous memory, the memory element acts as a latch and you must check the setup and hold time for the latch. An example is shown in [Figure 5–14](#). The longest clock path from `clk6` to destination memory is 9.251 ns. The shortest clock path from `clk6` to source register is 2.302 ns. Thus the largest clock skew is 6.949 ns. The shortest register to memory delay is 3.503 ns and the micro hold delay of the destination register is 0.106 ns. As a result, the clock skew is longer than the data path and the circuit does not operate normally.

Figure 5–14. Clock Skew

Multiple Clock Domains

When compiling designs that use more than one clock, the Quartus II software analyzes timing for register-to-register paths controlled by the different clocks, and reports the slack results. The Timing Analyzer disregards any paths between unrelated clock domains by default. See “[Cut Paths Between Unrelated Clock Domains](#)” on page 5–30 for more information.

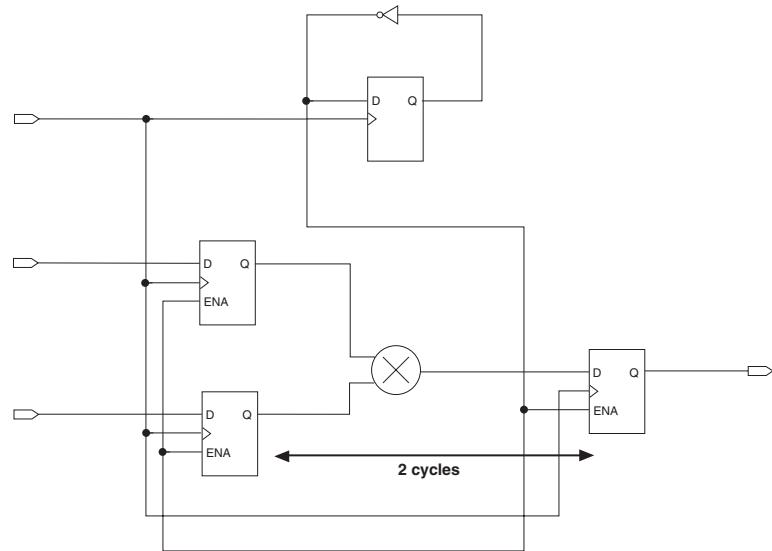
Correct multi-clock timing analysis involves the following steps:

1. Specify a desired f_{MAX} or clock period for an absolute clock. Define other clocks and their relationships, if any, to the absolute clock.
2. Assign the defined clock settings to the pins that supply the design's clock signals.
3. Compile the design. The Quartus II Timing Analyzer automatically reports circuit operability during compilation.

Multicycle Paths

Multicycle paths are data paths between registers that require more than one clock cycle to latch data at the destination register. For example, a register may need to trigger a signal on every second or third rising clock edge. [Figure 5–15](#) shows an example of a multicycle path between a multiplier's input registers and its output register.

Figure 5–15. Example Diagram of a Multicycle Path

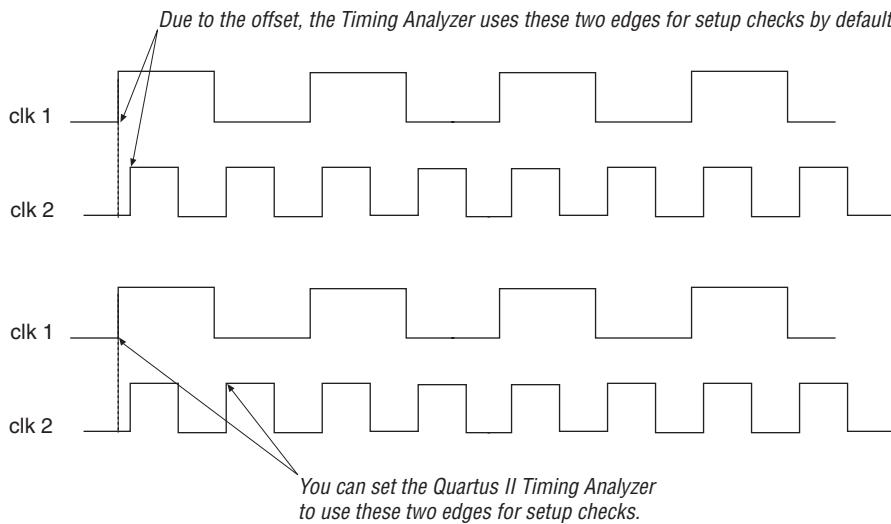


Multicycle Assignment

Make **Multicycle** assignments to specify the number of clock cycles required before a register should latch a value. **Multicycle** assignments delay the latch edge, relaxing the required setup relationship. You can assign multicycle paths in your designs to instruct the Quartus II Timing Analyzer to relax its measurements, thus avoiding incorrect setup or hold time violation reports.

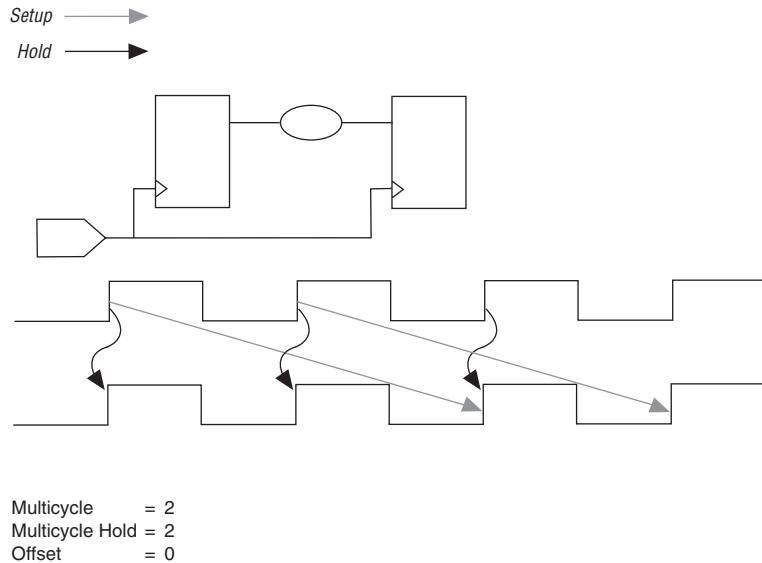
Figure 5–16 shows a timing diagram for a multicycle path that exists in a design with related clocks, with a small offset between the clocks.

Figure 5–16. Multicycle Paths with Offset Between Clocks



Multicycle Hold Assignment

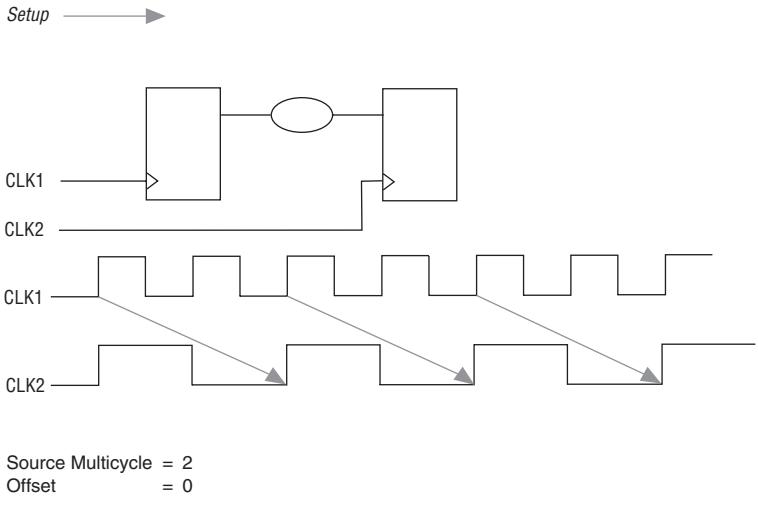
A **Multicycle Hold** assignment, shown in Figure 5–17, specifies the minimum number of clock cycles required before a register should latch a value. If no **Multicycle Hold** value is specified, the **Multicycle Hold** value defaults to the value of the **Multicycle** assignment.

Figure 5–17. Multicycle Hold Assignment

Source Multicycle Assignment

Source Multicycle assignments are used to extend the required delay by adding periods of the source clock rather than the destination clock.

Source Multicycle assignments are useful when the source and destination registers are clocked by related clocks at different frequencies.

Figure 5–18. Source Multicycle Assignment

Source Multicycle Hold Assignment

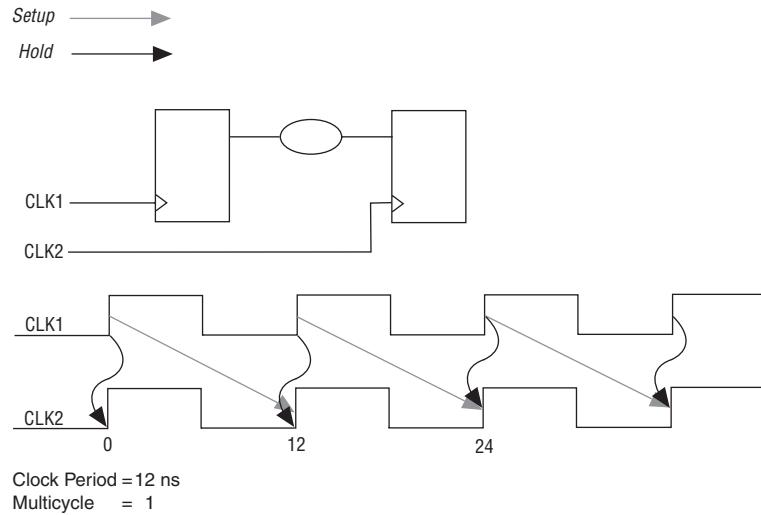
The **Source Multicycle Hold** assignment is useful when the source and destination registers are clocked by related clocks at different frequencies. This assignment allows you to increase the required hold delay by adding source clock cycles.

Typical Applications of Multicycle Assignments

The following examples describe how to use multicycle assignments in your designs.

Simple Multicycle Paths

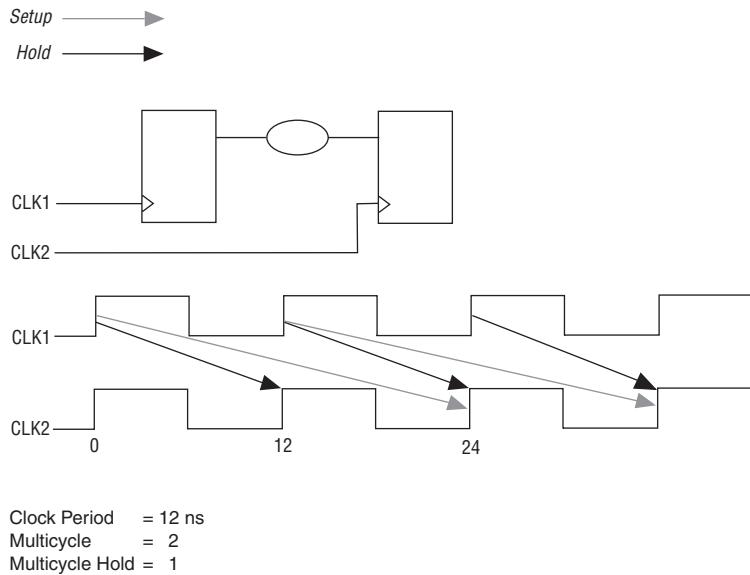
Figure 5–19 shows the measurement of t_{SU} and t_H for a standard path with a multicycle of 1.

Figure 5–19. t_{SU} & t_H Standard Measurement Paths

In the example shown in Figure 5–19, both `clk1` and `clk2` have the same period and zero offset. In this figure, where the clocks have a period of 12 ns, the data delay between the source and destination registers must be between 0 ns and 12 ns in order for the circuit to operate. If the data delay is longer than one clock period and the circuit is intended to operate as a Multicycle circuit, you must add a **Multicycle** assignment of 2.

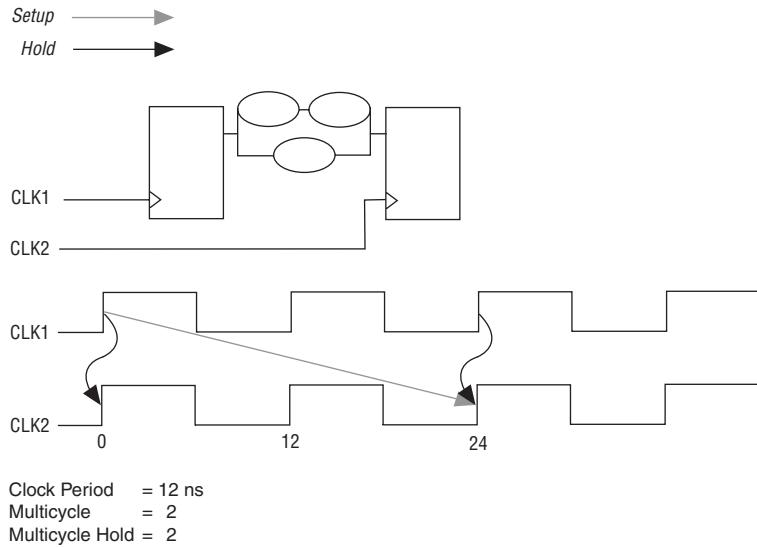


When you make **Multicycle** or **Source Multicycle** assignments, the Timing Analyzer sets the **Default Multicycle Hold** setting to the value of the **Multicycle** setting.

Figure 5–20. Timing Analysis

In Figure 5–20, the data delay between the two registers is longer than one clock cycle, but is less than two clock cycles. This circuit requires two clock cycles for a change at the input of the source register to appear at the destination register. The t_{SU} check on $clk2$ is performed at the second clock period (at 24 ns) and the t_H check is performed at the next period (at 12 ns). This analysis ensures that the data delay is between 12 ns and 24 ns. The minimum data delay is 12 ns and the maximum delay is 24 ns.

Figure 5–21 illustrates a design that has two data paths between the registers. One data delay is shorter than one clock period and the other data delay is longer than one clock period but shorter than two clock periods. The circuit is intended to operate as a multicycle path.

Figure 5–21. Data Path Delay Example

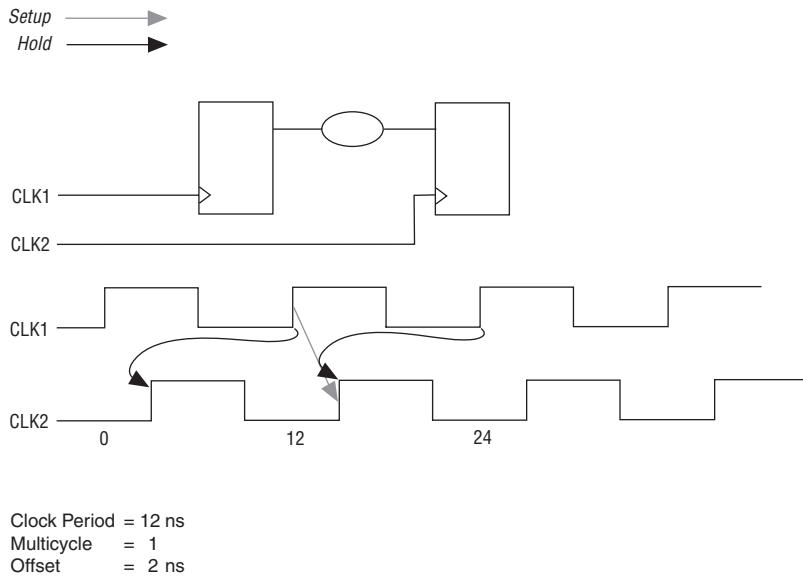
In **Figure 5–21**, the circuit is intended to operate with a multicycle path of two, however one of the data paths between the registers is less than one clock cycle.

t_{SU} is measured at the second clock edge and t_H is measured on the launch edge. The data delay must be between 0 ns and 24 ns for the circuit to operate.

Multicycle Paths with Offsets

In the example shown in [Figure 5–22](#), clk2 is offset from clk1 by 2 ns.

Figure 5–22. Multicycle Paths with Offsets



The setup time for clk2 is 2 ns and the hold time is –10 ns. Therefore the data delay must be between –10 ns and 2 ns. It is unlikely that the design is intended to latch the data within 2 ns, but it is probably intended to latch the data on the second clk2 edge, that is, operate as a multicycle path of two. If you set a **Multicycle** of 2 and **Multicycle Hold** assignment of 1, the setup requirement is 14 ns and the hold requirement is 2 ns, as shown in [Figure 5–23](#). The circuit operates as a multicycle path of two, assuming the data delay between the registers is between 2 ns and 14 ns.

The following Tcl commands are used to specify the multi-cycle assignments shown in [Figure 5–23](#):

```
set_multicycle_assignment -setup -from clk1 -to clk2 -end 2
set_multicycle_assignment -hold -from clk1 -to clk2 -end 1
```

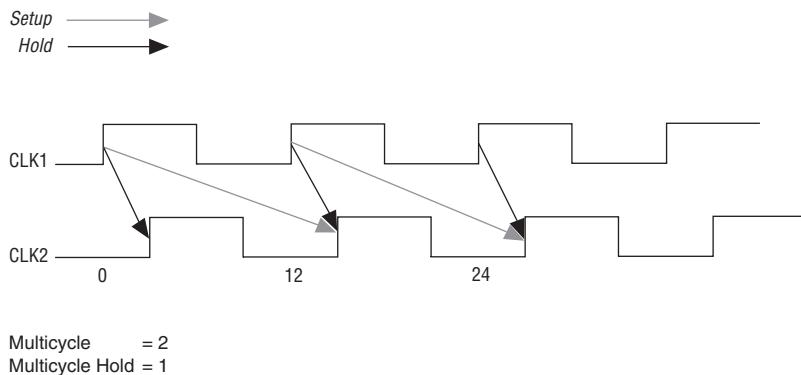
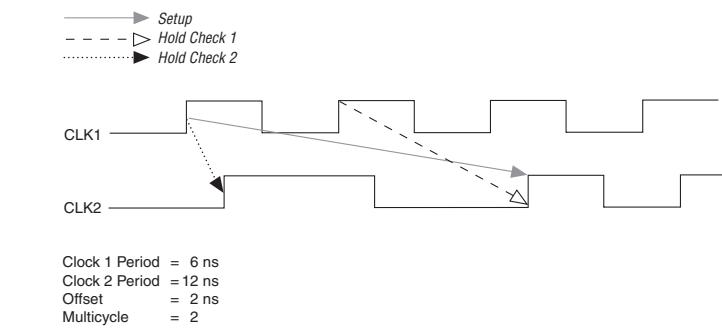
Figure 5–23. Hold Requirements***Multicycle Paths Across Multi-Frequency Domains***

Figure 5–24 is a timing diagram representing data traveling from a fast clock domain to a slow clock domain with an offset between the clock edges. Since data is transferring from a fast clock domain to a slow clock domain, it must remain stable for at least two source clock cycles, otherwise the data is lost. Without a **Multicycle** assignment, the Timing Analyzer calculates a data setup requirement of 2 ns, the value of the offset between the two clocks. The **Multicycle** assignment of 2 relaxes the setup requirement by extending it to the next destination clock edge.

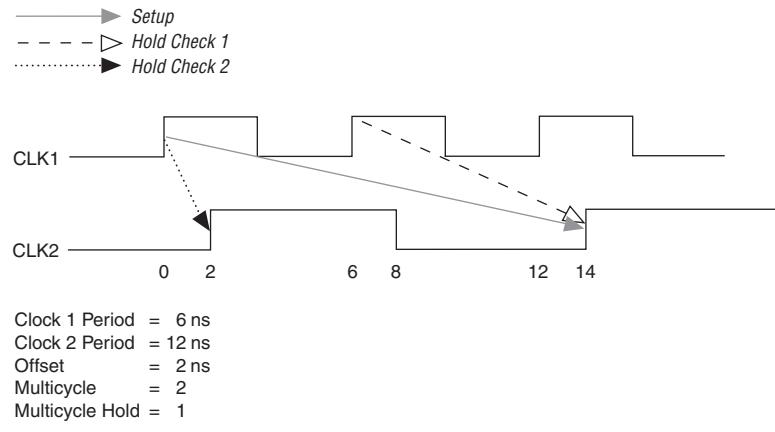
Figure 5–24. Multicycle Hold Checks

There are two hold relationships that the Timing Analyzer checks for multicycle paths in multi-frequency clock domain analysis. One check ensures that data clocked out of the source register after the launch edge is not latched by the destination register. This is illustrated by the dashed

line in [Figure 5–24](#). The other check ensures that data is not captured at the destination by the clock edge before the latch edge. This is illustrated by the dotted line in [Figure 5–24](#).

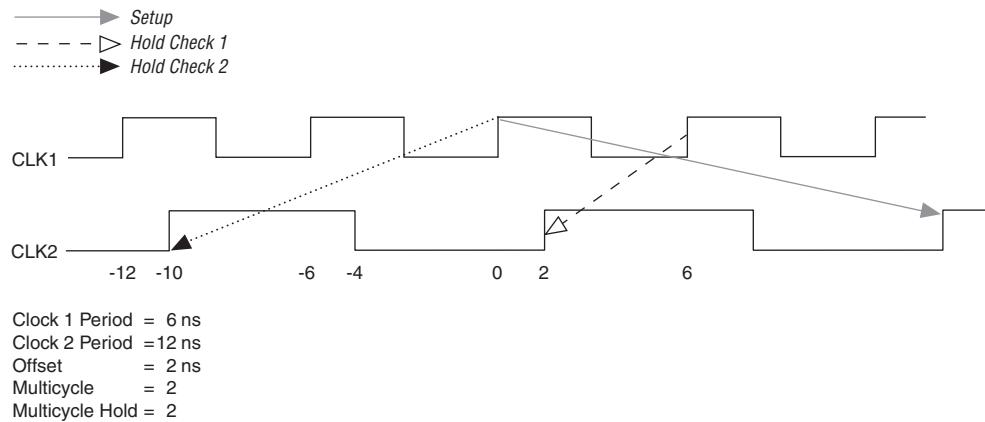
[Figure 5–25](#) illustrates hold time checks for a **Multicycle Hold** assignment of 1.

Figure 5–25. Multicycle Hold of 1



The first check, illustrated with the dashed line, requires a minimum data delay of 8 ns (14 ns - 6 ns). The second check, illustrated with the dotted line, requires a minimum data delay of 2 ns (2 ns - 0 ns). Data must have a maximum delay of 14 ns and a minimum delay of 8 ns to meet the **Multicycle** and **Multicycle Hold** requirements.

[Figure 5–26](#) illustrates hold time checks for the **Default Multicycle Hold** value of 2.

Figure 5–26. Multicycle Hold of 2

The **Multicycle Hold** value of 2 relaxes the hold time requirement by moving the reference edge one destination clock cycle earlier for the hold time calculation. The first check, illustrated with the dashed line, requires a minimum data delay of -4 ns (2 ns – 6 ns). The second check, illustrated with the dotted line, requires a minimum data delay of -10 ns (0 – 10 ns). Data must have a maximum delay of 14 ns and a minimum delay of -4 ns to meet the **Multicycle** and **Multicycle Hold** requirements.

Figure 5–27 is a timing diagram representing data going from a slow clock domain to a fast clock domain with an offset between the clock edges. The **Multicycle** assignment of 4 relaxes the setup requirement by extending it to the fourth destination clock edge, but the hold requirement is unchanged.

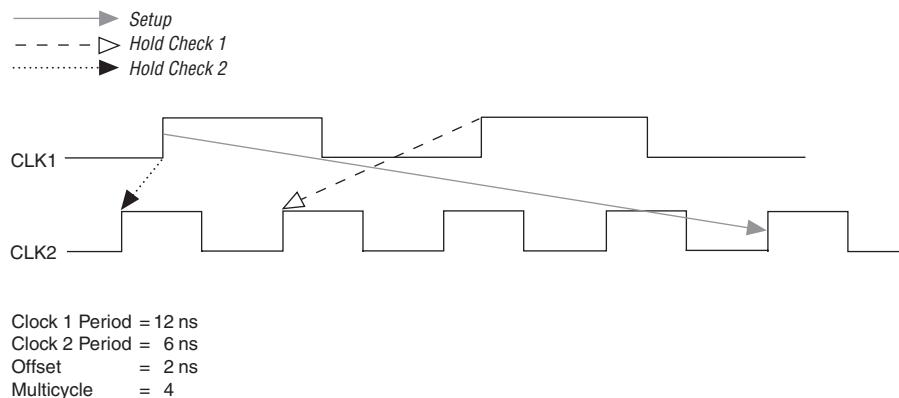
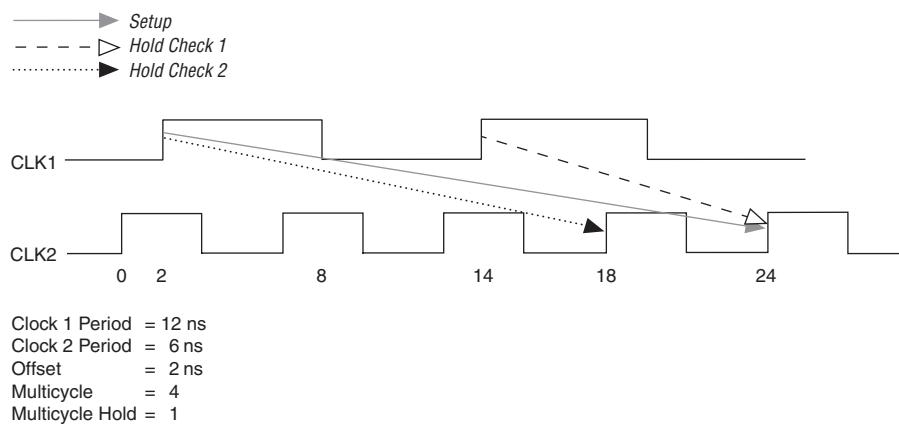
Figure 5–27. Multicycle Hold Checks

Figure 5–28 illustrates hold time checks for a **Multicycle Hold** assignment of 1.

Figure 5–28. Multicycle Hold of 1

The first check, illustrated with the dashed line, requires a minimum data delay of 10 ns (24 ns – 14 ns). The second check, illustrated with the dotted line, requires a minimum data delay of 16 ns (18 ns – 2 ns). Data must have a maximum delay of 22 ns and a minimum delay of 16 ns to meet the **Multicycle** and **Multicycle Hold** requirements.

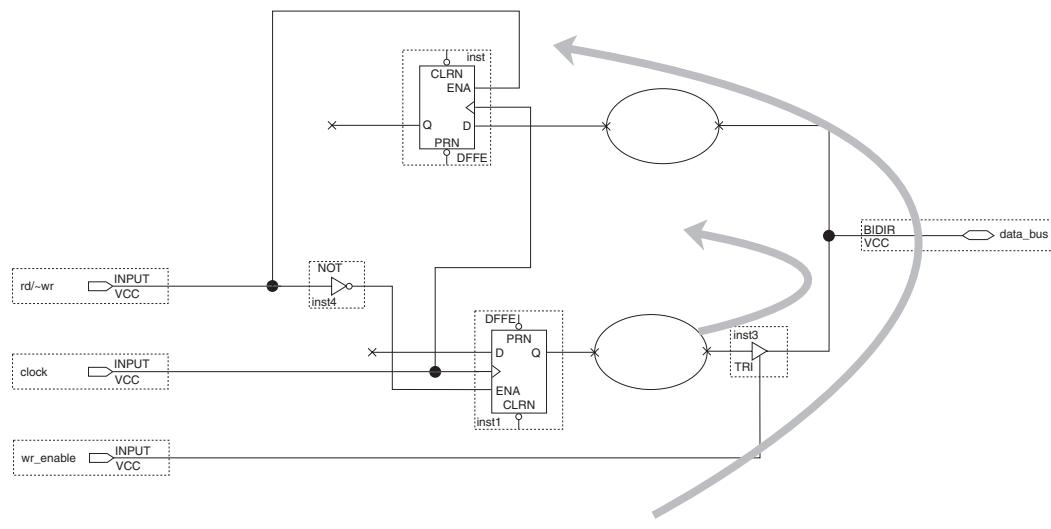
False Paths

A false path is any path that is not relevant to a circuit's operation. You can make a variety of assignments to exclude false paths from timing analysis. Global assignments excluding common false paths are turned on in the **Timing Requirements & Options** page of the **Settings** dialog box by default. You can make separate **Cut Timing Path** assignments to cut individual false paths.

Cut Off Feedback from I/O Pins

This option, which is on by default, cuts off feedback paths from I/O pins as shown in [Figure 5–29](#).

Figure 5–29. Cut Off Feedback from I/O Pins

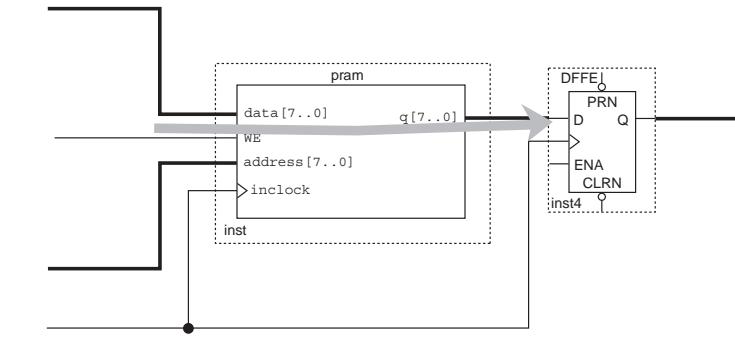


The paths marked with arrows are not measured by timing analysis when this option is turned on. Turn off **Cut off feedback from I/O pins** to measure these paths during timing analysis.

Cut Off Read During Write Signal Paths

This option is turned on by default and cuts the path from the write enable register through the embedded system block (ESB) to a destination register, as shown in Figure 5–30.

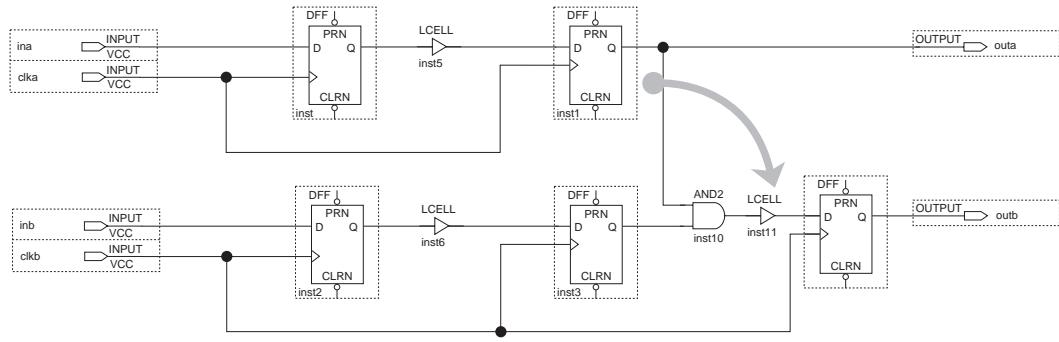
Figure 5–30. Cut Off Read During Write Signal Paths



The path marked with an arrow between the WE input to the memory block pram and the register inst4 is not reported by the Timing Analyzer. This path is reported if **Cut off read during write signal paths** is turned off.

Cut Paths Between Unrelated Clock Domains

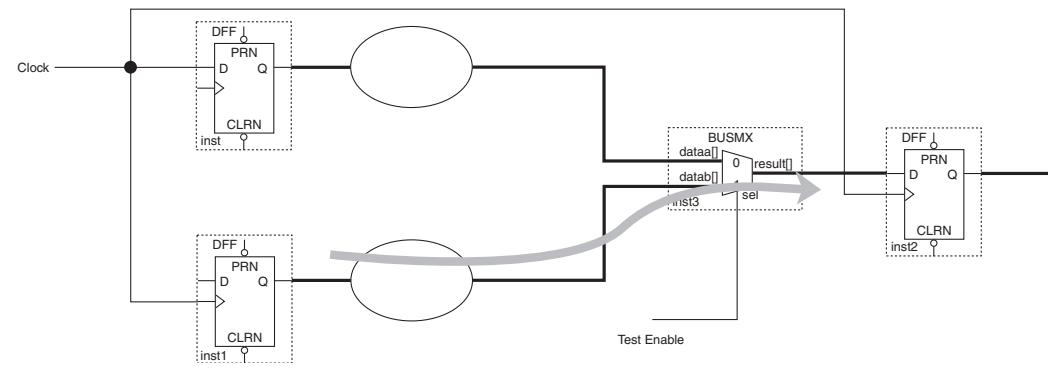
By default, the Quartus II software cuts paths between unrelated clock domains when there are no timing requirements set or only the default required f_{MAX} is specified. This option cuts paths between unrelated clock domains if individual clock assignments are set but there is no defined relationship between the clock assignments. See Figure 5–31.

Figure 5–31. Cut Paths Between Unrelated Clock Domains

For the circuit shown in Figure 5–31, the path between inst1 and inst4 is not measured or reported by the Timing Analyzer. If you turn off **Cut timing paths between unrelated clock domains**, the Timing Analyzer includes these paths as part of timing analysis.

Cut Timing Path

You can make **Cut Timing Path** assignments to paths that are not used under normal operation, such as paths through test logic. Figure 5–32 shows an example of a false path.

Figure 5–32. False Path Signal

In Figure 5–32, the path from inst1 through the multiplexer to inst2 is used only for design testing. This false path is not used under normal operation and should not be considered during timing analysis. You can remove a false path from timing analysis with a **Cut Timing Path** assignment from register inst1 to register inst2.

Fixing Hold Time Violations

Hold time violations usually occur when clock skew is greater than data delay between two registers. Clock skew between registers can occur if you use gated clocks in your design. It can also occur if some clocks are inferred from flip-flops or other logic. You can use any of the following guidelines to address reported hold time violations.

Make Multicycle Hold Assignments

Depending on your design functionality, you can relax the hold relationship with **Multicycle Hold** or **Source Multicycle Hold** assignments.

Reduce Clock Skew

Using global buffers for clock distribution minimizes clock skew, but these buffers do not necessarily provide the shortest delay path. You can route gated clocks using non-global buffers to access faster clock trees, because the skew is already caused by the clock-gating logic. You can also use a PLL to divide a clock signal instead of using other logic which may cause clock skew. Because gated clocks are common causes of clock skew, Use clock enables instead of gated clocks in your design where possible.

Increase Data Delay

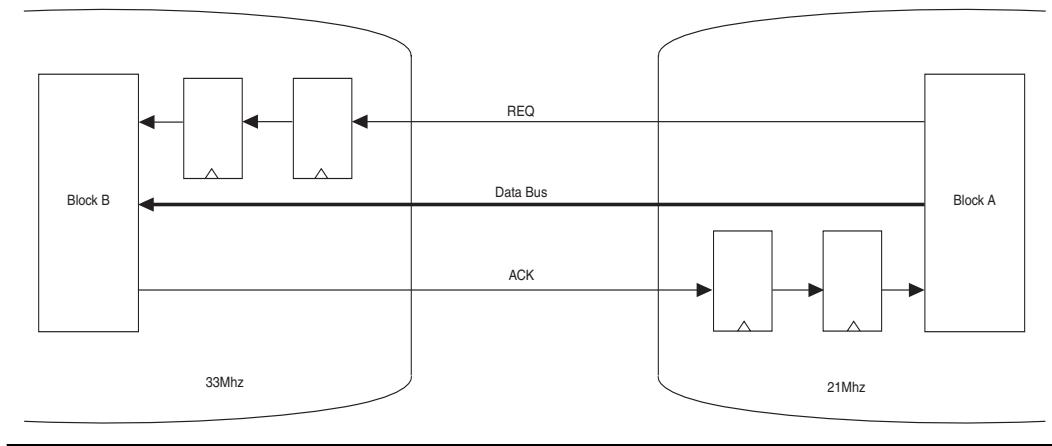
You can increase data delay until it is greater than clock skew to resolve hold time violations. One way to do this is with the **Logic Cell Insertion** assignment. You can specify a number of LCELL primitives to automatically insert in the failing path. These primitives do not change the functionality of your design. Another way to increase data delay is to assign nodes to LogicLock regions in separate areas of the device. This increases the routing delay along the path.

Timing Analysis Across Asynchronous Domains

In cases where source and destination clocks are unrelated, timing analysis across unrelated clock domains is not very useful because cross-domain paths are asynchronous. You can make **Cut Timing Path** assignments to cross-domain paths and use special design techniques to make sure that asynchronous signals do not cause metastability. One of the most common techniques used is to enforce a full handshake protocol between the asynchronous boundaries. in [Figure 5-33](#), Block A asserts the REQ signal when data is ready. Block B synchronizes the REQ signal through two flip-flops and then asserts the ACK signal when it has latched the data. Block A synchronizes the ACK signal through two flip flops and

then de-asserts the REQ signal. This technique guarantees that the data is transferred correctly and there is no metastability caused by asynchronous signals.

Figure 5–33. Interaction Across Asynchronous Boundaries



Best Case Timing Analysis

Best case timing analysis (formerly known as minimum timing analysis) measures and reports minimum t_{CO} , minimum t_{PD} , t_H , and clock hold. Best case timing analysis is performed by checking for minimum delay requirements with best-case (delay) timing models. Best-case timing models characterize device operation at the highest voltage, fastest process and lowest temperature conditions. Worst-case timing models (delay models) characterize device operation based on the slowest process, lowest voltage, and highest temperature conditions. Minimum delay checks, like t_H , are also reported during regular timing analysis using worst-case delay models.

Best Case Timing Analysis Settings

You can make global minimum t_H , minimum t_{CO} , and minimum t_{PD} assignments in the **Minimum Delay Requirements** section of the **Timing Requirements & Options** page of the **Settings** dialog box (Assignments menu). You can also make individual minimum timing settings to pins and registers in your design.

Performing Best Case Timing Analysis

To perform best case timing analysis with the best-case timing models (delay models), choose **Start > Start Timing Analyzer (Fast Timing Model)** (Processing menu). If you use the **quartus_tan** executable, specify the **--fast_model=on** option as shown below:

```
quartus_tan <project_name> --fast_model=on ↵
```

Best Case Timing Analysis Reporting

You can examine the results of best case timing analysis in the Timing section of the **Compilation Report** (Processing menu) in the Quartus II GUI. The text-based report generated during timing analysis is called **<project name>.tan.rpt**.



The Quartus II Timing Analyzer saves both minimum and regular timing analysis results in **<project name>.tan.rpt**, potentially overwriting previous timing analysis results. To preserve a copy of your results, save the file with a new name before the next compilation or analysis.

Even when you perform regular, worst-case timing analysis, there can be reports in the Timing Analysis section of the Compilation Report listing minimum delay checks. These results are generated by reporting the minimum delay checks using the worst-case timing models (delay models).

Recovery and Removal Analysis

The Enable Recovery/Removal analysis option reports the results of setup and hold checks for paths that have an asynchronous clear, preset, or load signal. This option can be found in the More Timing Settings dialog box.

With this option enabled, the Quartus II Timing Analysis engine generates a report file for the recovery analysis and the removal analysis.

Recovery Report

When you turn on the Enable Recovery/Removal analysis option, the Timing Analyzer determines the minimum amount of time required between an asynchronous control signal going inactive and the next active clock edge, compares this to your design, and reports the results as slack following timing analysis. The Recovery report alerts you to a condition in which an active clock edge occurs too soon after the asynchronous input goes inactive, thus rendering the data uncertain.

Figure 5–34. Recovery Report

Recovery: 'sys_clk'						
	Slack	From	To	From Clock	To Clock	Required Time
						Actual Time
1	2.182 ns	areset	inst5 sys_clk	sys_clk	sys_clk	7.921 ns
2	2.672 ns	areset_reg	inst6 sys_clk	sys_clk	sys_clk	8.445 ns
						5.773 ns



An input maximum delay assignment must be made to a pin for the Quartus II timing analyzer to perform a recovery analysis.

Removal Report

When you turn on the Enable Removal/Removal analysis option, the Timing Analyzer determines the minimum amount of time required between a clock edge that occurs while an asynchronous input is active, and the removal of the asynchronous control signal. The Timing Analyzer then compares this to your design, and reports the results as slack following timing analysis. 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 data uncertain.

Figure 5–35. Removal Report

Compilation Report

Removal: 'sys_clk'

	Slack	From	To	From Clock	To Clock	Required Time	Actual Time
1	1.996 ns	areset_reg	inst6	sys_clk	sys_clk	3.777 ns	5.773 ns
2	2.620 ns	areset	inst5	sys_clk	sys_clk	3.119 ns	5.739 ns



An input maximum delay assignment must be made to a pin for the Quartus II timing analyzer to perform a removal analysis.

Early Timing Estimation

The majority of Quartus II software compilation time is consumed by the place-and-route algorithms used to obtain optimal design results. To accelerate the design process for large designs, the Quartus II software provides The Early Timing Estimate feature. Choosing **Start > Start Early Timing Estimate** (Processing menu) provides a quick static timing analysis in a fraction of the time required for a full compilation. The time reduction is possible because the **Early Timing Estimate** option does the following:

- Places and routes a design up to ten times faster than when fully fitting a design
- Generates a full-timing report based on estimated design delays



An Early Timing Estimate fit is not fully optimized or legally routed. The timing report is only an estimate. Typically, the estimated delays are within 20% of those obtained with a full compilation.

The **Early Timing Estimate** page of the **Settings** dialog box (Assignments menu) provides three settings for generating timing estimates. [Table 5–2](#) describes the three available settings.

Table 5–2. Early Timing Estimate Setting Options	
Setting	Description
Realistic (default setting: estimates final timing using standard fitting)	Estimates delays that will likely be close to a full compilation's results (default value).
Optimistic (estimates best case final timing)	Estimates delays that are lower than those likely to be achieved by a full compilation. This makes the estimate of performance optimistic.
Pessimistic (estimates worst case final timing)	Estimates delays that are higher than those likely to be achieved by a full compilation. This makes the estimate of performance pessimistic.



The settings **Realistic**, **Optimistic**, and **Pessimistic** each require the same compile time when set.

To use the Early Timing Estimate feature, choose **Start Early Timing Estimate** (Processing menu) after you have synthesized your design in the Quartus II software.

Latch Analysis

The Quartus II Timing Analyzer performs a conservative static timing analysis as part of every full compilation. This conservative static timing analysis applies stringent requirements to all design timing nodes, which ensures that your design functions in the targeted device.

These stringent requirements direct the Timing Analyzer to analyze latches as synchronous elements, rather than as combinational elements. Although latches are implemented as a look-up-tables (LUT) feeding back onto themselves, the Early Timing Estimate feature directs the Timing Analyzer to analyze all latches as synchronous elements. Specifically, the clock enable is analyzed as an inverted clock. Also, the timing analyzer reports the results of setup and hold analysis on these latches.

Scripting Support

You can make timing assignments, perform timing analysis, and generate reports in the Quartus II software GUI or with Tcl commands. You can use simple Tcl commands to generate customized timing reports, and you can write scripts with advanced timing analysis commands to traverse timing netlist and report results. You can use the command-based timing analyzer in an interactive shell mode where you can run timing analysis Tcl scripts.

General Timing Analyzer Commands

To run the timing analyzer in interactive shell mode, type the following command at a command prompt:

```
quartus_tan -s ↵
```

To run a Tcl script, type the following command at a command prompt:

```
quartus_tan -t <tcl_file> ↵
```

You frequently need the following commands when running timing-related scripts:

```
package require ::quartus::<package_name>
```

```
load_package <package_name> -version <version_number>
```



To access the advanced Tcl functions for traversing the timing netlist, use the `advanced_timing 1.1` package; to make project-wide assignments, use the `project 3.0` package.

To open the project in the project directory for analyzing or making timing assignments, type the following command at a command prompt:

```
project_open <project_name> ↵
```

Creating a Timing Netlist

Timing netlists are created in memory and used for analysis and reporting commands. To generate a timing netlist, use the `create_timing_netlist` command in your scripts.

When creating a timing netlist, the `-min` option can be used to generate delays based on best-case operating conditions instead of the default worst-case delays. Use `-skip_dat` if you have previously run the delay annotator.

To conserve memory, use the `delete_timing_netlist` when the netlist is no longer needed.



Always end Quartus II Tcl scripts with `project_close`. The `project_close` command closes the project and writes out all of the assignments to the QSF unless the `-dont_export_assignments` option is used.

Assignments created, or modified, after opening a project are not committed to the QSF unless `project_close` or `export_assignments` is used.



You can view the list of available Tcl packages and help for each command by launching the *Quartus II Command-Line and Tcl API Help* browser from a command line or the Quartus II Tcl Console, using the command `quartus_sh --qhelp ↵`.

Advanced Timing Analysis & Reports

The `report_timing` command gives you more control over how you want to report your timing analysis results. The following shows correct usage for the `report_timing` command:

```
report_timing [-reuse_delays] [-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] [-dqqs_read_capture] [-stdout]
[-file <name>] [-append] [-from <names>] [-to <names>]
[-clock_filter <names>] [-longest_paths]
[-shortest_paths] [-all_failures]
```

The following command writes out worst timing path, one for each of the t_{SU}, t_H, t_{CO} , minimum t_{CO} , clock setup and clock hold timing reports based on worst-case delay models into a text file called **file_name**:

```
report_timing -file file_name
```

The following command writes out 2 timing paths for each of the constraints in **file_name**:

```
report_timing -npaths 2 -file file_name
```

The following command reports the 3 worst paths of t_{su} constraint:

```
report_timing -tsu -npaths 3
```

The following command reports one timing path per constraint related to clock domains whose names end with _clk0 only. The filtering can be further restricted by using more descriptive string matching such as *pll0*_clk0. These clock names are not limited to absolute or relative clocks defined by the user but can also include outputs of any PLL usage in the design.

```
report_timing -clock_filter *_clk0
```

The following command lists all timing paths starting from input, in1, to any registers or outputs that have utopia as part of their name.

```
report_timing -from in1 -to *utopia*
```

The following command lists all timing paths that end at bit 4 of the output bus out [4 : 0]. Precede every bracket character with a backslash and enclose the string in braces for proper interpretation:

```
report_timing -to {out\[4\]}
```

The following script reports all the registers in a design along with their respective locations on the chip.

```
package require ::quartus::advanced_timing
project_open <project_name>
create_timing_netlist
create_p2p_delays
foreach_in_collection node [get_timing_nodes -type reg] {
    set reg_name [get_timing_node_info -info name $node]
    set location [get_timing_node_info -info location $node]
    puts "register: $reg_name location: $location "
}
project_close
```

The following script begins by traversing through a list of all the registers in a design by using the get_timing_nodes -type reg command. The script then uses a foreach loop to trace the clock path back to the input clock pin. Using this technique, the total clock insertion delay for each register is computed from the input reference clock pin, including the PLL offset. At the end, each register name, its associated clock name, and the total clock network delay from the input clock pin for each register is printed out. Being able to print out clock insertion delays for each register in the design helps figure out minimum and maximum clock skews between different clock domains even when more than one PLLs are involved.

```
package require ::quartus::advanced_timing

proc split_time { a } {
    set pieces [split $a]
```

```
if {[string equal $ps [lindex $pieces 1]]} {
    set time [expr 1000 * [lindex $pieces 0]]
} else {
    set time [lindex $pieces 0]
}
return $time
}
project_open <project_name>
create_timing_netlist
create_p2p_delays
foreach_in_collection node [get_timing_nodes -type reg] {

    set reg_name [get_timing_node_info -info name $node]
    set delays_from_clock_list [get_delays_from_clocks $node]
    set delays_from_clock [lindex $delays_from_clock_list 0]
    set clock_node_id [lindex $delays_from_clock 0]
    set fanin [get_timing_node_fanin -type clock $clock_node_id]
    set pll_delay_list [lindex $fanin 0]
    set pin_to_pll_list [lindex [get_timing_node_fanin -type clock \
        [lindex $pll_delay_list 0]] 0]

    set sum_of_delays [expr [split_time [lindex $pll_delay_list 1]] + \
        [split_time [lindex $pll_delay_list 2]] + [split_time \
            [lindex $pin_to_pll_list 2]]]
    set clock_name [get_timing_node_info -info name \
        [lindex $delays_from_clock 0]]
    set longest [lindex $delays_from_clock 1]
    set shortest [lindex $delays_from_clock 2]

    puts "-> clock is $clock_name"
    puts "-> register name $reg_name"

puts "-> total clock pin to reg delay [expr {$sum_of_delays + [split_time $longest]}] ns"
}
project_close
```

Conclusion

Evolving design and aggressive process technologies require larger and higher-performance FPGA designs. Increasing design complexity demands enhanced timing analysis tools that aid designers in verifying design timing requirements. Without advanced timing analysis tools, you risk circuit failure in complex designs. The Quartus II Timing Analyzer incorporates a set of powerful timing analysis features that are critical in enabling system-on-a-programmable-chip designs.

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 chapters:

- [Chapter 6, Synopsys PrimeTime Support](#)
- [Chapter 7, PowerPlay Early Power Estimator](#)

Revision History

Chapter *Simulation-Based Power Estimation* was removed from the *Quartus II Handbook*.

The table below shows the revision history for [Chapter 6](#) and [7](#).

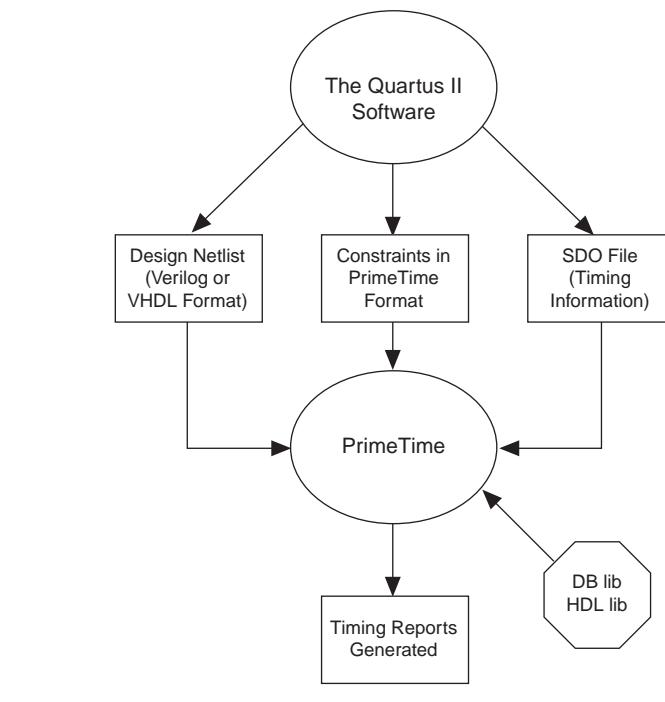
Chapter(s)	Date / Version	Changes Made
6	Dec 2004 v2.1	<ul style="list-style-type: none">● Chapter 6 was formerly Chapter 5.● Minor changes to tables and figures.
	June 2004 v2.0	No changes to document.
	Feb 2004 v1.0	Initial release.
7	Dec. 2004 v2.1	<ul style="list-style-type: none">● Chapter 7 was formerly Chapter 6.● Updates to information, tables, and figures.● New functionality for Quartus II software 4.2.● New figures.
	June 2004 v2.0	<ul style="list-style-type: none">● Updates to tables, figures.● New functionality for Quartus II software 4.1.
	Feb. 2004 v1.0	Initial release.

qii53005-2.1

Introduction

PrimeTime is an industry standard sign-off tool used to perform 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 PrimeTime format), and libraries to the PrimeTime environment. [Figure 6–1](#) shows the PrimeTime flow diagram.

Figure 6–1. PrimeTime Flow Diagram



This chapter describes the following:

- How to make the settings in the Quartus II software to cause it to generate files for use by the PrimeTime software
- The contents and intended use of the files generated by the Quartus II software
- How to run the PrimeTime software

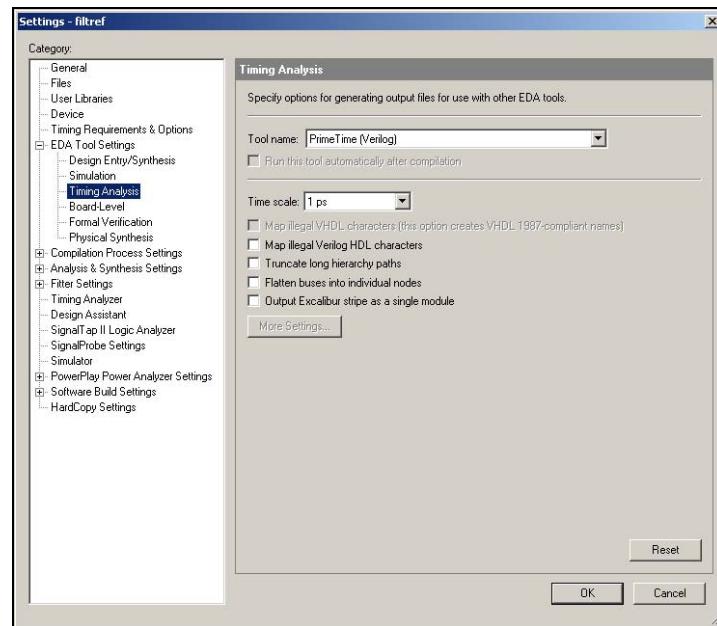
- How to read the output of the PrimeTime software

Quartus II Settings for Generating PrimeTime Files

To set the Quartus II software to generate files for the PrimeTime software, perform the following steps:

1. In the Quartus II software, choose **Settings** (Assignments menu).
2. Choose **EDA Tool Settings > Timing Analysis** under **Category** to display the **Timing Analysis** page of the **Settings** dialog box.
3. In the **Tool name** list, select **PrimeTime (Verilog)** or **PrimeTime (VHDL)**, depending on the HDL language used in your Quartus II project (Figure 6–2).

Figure 6–2. Setting the Quartus II Software to Generate PrimeTime Files



This setting enables the Quartus II software to produce three files for the PrimeTime tool, which are saved in the `<project directory>/timing/primetime`, where `<project directory>` is the location of your Quartus II project.

Files Generated for the PrimeTime Environment

The Quartus II software generates a flat netlist, a Standard Delay Output (.sdo) file, and a Tcl script to run in the PrimeTime software that triggers timing analysis of your Quartus II project. These files are saved in the `<project directory>/timing/primetime` directory, where `<project directory>` is the location of your Quartus II project.

The Netlist

The netlist is written and saved as either `<project name>.vo` or `<project name>.vho`, depending on whether **PrimeTime (Verilog)** or **PrimeTime (VHDL)** is selected in the **Tool name** list on the **Timing Analysis** page of the **Settings** dialog box (Assignments menu). This file contains the flat netlist representing the entire design.

The Standard Delay Output File

The Quartus II software saves the SDO file as either `<project_name>.v.sdo` or `<project_name>.vhd.sdo`, depending on whether **PrimeTime (Verilog)** or **PrimeTime (VHDL)** is selected in the **Tool name** list on the **Timing Analysis** page of the **Settings** dialog box (Assignments menu).

This file contains the timing information for each timing arc in the design. The Quartus II software generates this file using worst-case delay values for the timing arcs. The Quartus II Timing Analyzer uses these worst-case timing models by default, to give conservative results.

To generate the SDO using best-case timing analysis, perform the following steps:

1. In the Quartus II software, choose **Start > Start Timing Analyzer (Fast Timing Model)** (Processing menu).
2. After best-case timing analysis is complete, choose **Start > Start EDA Netlist Writer** (Processing menu) to create a `<project_name>.v_min.sdo` or `<project_name>.vhd_min.sdo` file, which contains the best-case delay values for each timing arc.



If you are using best-case timing models, the Tcl script generated by the Quartus II software for use in the PrimeTime software must be edited so that the files generated by the Quartus II EDA Netlist Writer from minimum timing analysis are used instead of those generated by default.

The Tcl Script

The Quartus II software saves a Tcl script as either `<project_name>.pt_v.tcl` or `<project_name>.pt_vhd.tcl`, depending on whether **PrimeTime (Verilog)** or **PrimeTime (VHDL)** is selected in the **Tool name** list on the **Timing Analysis** page of the **Settings** dialog box (Assignments menu).

This script specifies the search path to, and the names of, the PrimeTime database library files provided with the Quartus II software. The search path and link path are defined at the beginning of the Tcl file. The link path is a space-delimited list containing the names of all database files used by the PrimeTime software.



The script also directs the PrimeTime software to refer to `<device family>.all_pt.v` or `<device family>.all_pt.vhd`, which contains the Verilog/VHDL description of each library cell for the targeted device family.

Here is an example of the search path and link path defined in the Tcl script:

```
set quartus_root "/apps1/altera/quartus/II-4.2/"
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_pll_lib.db stratixii_dll_lib.db alt_vt1.db]

read_verilog stratixii_all_pt.v
```

This Tcl script contains constraints in PrimeTime format that are converted by the Quartus II software from constraints made in the project. This Tcl script also includes a PrimeTime command to read the SDO file generated by the Quartus II software. If you are using best-case timing analysis, specify the correct SDO file name. You can place additional PrimeTime commands in the Tcl script to report on, or analyze, timing paths.

Sample of Constraints Specified in PrimeTime Format

The PrimeTime constraints shown in [Table 6-1](#) are automatically generated by the Quartus II software. The `set_input_delay -max` command is equivalent to the `tSU` constraint in the Quartus II software.

Since `input_delay` in PrimeTime is defined as the data delay from clock edge to the input pin, and t_{SU} in the Quartus II software is the data delay from the input pin to clock edge, t_{SU} is subtracted from the clock period to calculate the `set_input_delay`. Table 6–1 shows the automatically-generated PrimeTime constraints and their Quartus II software equivalents.

Table 6–1. Equivalent Quartus II & PrimeTime Constraints	
Quartus II Equivalent	PrimeTime Constraint
Clock defined on input pin, clock of 10-ns period and 50% duty cycle	<code>create_clock -period 10.000 -waveform {0 5.000} \ [get_ports clk] -name clk</code>
t_{SU} of 1 ns on input pin, din	<code>set_input_delay -max -add_delay 9.000 -clock \ [get_clocks clk] [get_ports din]</code>
t_H of 1 ns on input pin, din	<code>set_input_delay -min -add_delay 1.000 -clock \ [get_clocks clk] [get_ports din]</code>
t_{CO} of 3 ns on output pin, out	<code>set_output_delay -max -add_delay 7.000 -clock \ [get_clocks clk] [get_ports out]</code>

Running the PrimeTime Software

The PrimeTime software runs on the UNIX operating system. The three files created by the Quartus II software are transferred to a UNIX workstation if they were not created there.

Analyzing Quartus II Projects

The PrimeTime software is controlled with Tcl scripts. You can run the `<project_name>_pt_v.tcl` script file, for example, by typing the following at a UNIX system command prompt:

```
pt_shell -f project_name_pt_v.tcl ↵
```

After all Tcl commands in the script are interpreted, the PrimeTime software returns control to the `pt_shell` prompt, where other commands can be used.

Other pt_shell Commands

More `pt_shell` commands are executed 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 ↵
```

List all commands available in the `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 activate `pt_shell` without a script file by typing `pt_shell ↵` at the UNIX command line prompt.

PrimeTime Timing Reports

To generate a list of the 100 worst paths, and place this data into a file called `file.timing`, type the following at the `pt_shell` prompt:

```
report_timing -nworst 100 > file.timing ↵
```

Timing paths in PrimeTime are listed in the order of most-negative-slack to most-positive-slack. Unlike Quartus II timing analysis reports, failing paths are not reported by the PrimeTime software under each constraint's category. Timing setup (t_{SU}) and timing hold (t_H) times are not listed separately. In PrimeTime, there is a start and end point given with each path to identify, 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.

```
report_timing -delay min ↵
```

This command can be used to create a minimum timing report or a list of hold-time-related violations. It is up to you to define what type of SDO file is being used. Both minimum delay and maximum delay SDO files can be generated from the Quartus II software.

Sample PrimeTime Timing Report

The following is a sample PrimeTime timing report. The start point in this report is a register clocked by clock, `clk`. The endpoint is an output pin, `out`. This is equivalent to either a t_{CO} or a Minimum t_{CO} path in the Quartus II software, depending on the `-delay` option. At the end of the report, "Violated" is listed, which means that the constraint was not met. A negative slack is also given, as in the Quartus II software.

```
Startpoint: ~I.out_reg
(rising edge-triggered flip-flop clocked by clk)
Endpoint: out (output port clocked by clk)
Path Group: clk
Path Type: max
Point                                         Incr    Path
clock clk (rise edge)                         0.00    0.00
clock network delay (propagated)              2.362   2.362
out~I.out_reg.clk (stratix_io_register)       0.00    2.362 r
out~I.out_reg.regout (stratix_io_register)     0.162*  2.524 r
```

out~I.out_mux3.M0 (mux21)	0.000	2.524	r
out~I.and2_22.Y (AND2)	0.000	2.524	r
out~I.out_mux1.M0 (mux21)	0.000	2.524	r
out~I.inst1.padio (stratix_asynch_io)	2.715H	5.239	r
out~I.padio (stratix_io)	0.000	5.239	r
out (out)	0.00	5.239	r
data arrival time		5.239	r
clock clk (rise edge)	10.000	10.000	
clock network delay (propagated)	0.000	10.000	
output external delay	-7.000	3.000	
data required time		3.000	
data required time		3.000	
data arrival time		-5.239	
slack (VIOLATED)		-2.239	

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, which can be customized to cause the PrimeTime software to produce reports of any violations and slacks.

qii53006-2.1

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. Stratix® II, Stratix, Stratix GX, Cyclone™, and MAX® II device power consumption can be estimated using the Microsoft Excel-based PowerPlay Early Power Estimator spreadsheet or analyzed with the PowerPlay Power Analyzer feature in the Quartus® II software, which is described in the *PowerPlay Power Analyzer* chapter in Volume 3 of the *Quartus II Handbook*.

You can use the PowerPlay Early Power Estimator spreadsheet, described in this chapter, during the board design and layout phase to obtain a power estimate and design for proper power management. The PowerPlay Power Analyzer feature in the Quartus II software is used to obtain a more accurate estimation of power after the design is complete. This allows you to ensure that thermal and supply budgets are not violated.

PowerPlay Early Power Estimator Spreadsheet



Before reading this chapter, you should be familiar with the PowerPlay Early Power Estimator spreadsheets for Stratix II, Stratix, Stratix GX, Cyclone, and MAX II device families that are available on the Altera® web site.

The PowerPlay Early Power Estimator spreadsheet, which provides a current (I_{CC}) and power (P) estimation is available on the Altera web site Stratix II, Stratix, Stratix GX, Cyclone, and MAX II device families, under **Design Utilities**. The Early Power Estimator spreadsheet is divided into sections, with each section representing an architectural feature of the device, such as the clock network, RAM blocks, and digital signal processing (DSP) blocks. You must enter the device resources, operating frequency, toggle rates, and other parameters in the Early Power Estimator spreadsheet to estimate the device power consumption. For given resource counts, the spreadsheet makes assumptions for routing and other details, based on typical customer designs. Based on these assumptions, the spreadsheet estimates the power consumption for resources. The sub-total of the I_{CC} and power consumed by each feature is reported in each section in millamps (mA) and milliwatts (mW), respectively.



For more information about how to use the Excel-based PowerPlay Early Power Estimator, see the *PowerPlay Early Power Estimator User Guide*.

Figures 7–1 through 7–5 show sections of the Stratix PowerPlay Early Power Estimator.

Figure 7–1. Device & I_{CC} Standby Sections in the Stratix PowerPlay Early Power Estimator

A	B	C	D	E	F	G
1						
2	ALTERA®	Altera® Stratix™ PowerPlay Early Power Estimator Version 3.0				
3	www.altera.com	Altera does not guarantee or imply the reliability, serviceability, or function of this Program or other items provided as part of this Program. The files contained herein are provided 'AS IS'. ALTERA DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.				
4		Copyright Altera Corporation. All rights reserved.				
5						
6	Comments:					
7						
8						
9						
10						
11	Device					
12	Device	Package	Temperature Grade	V _{CCINT}	Total P _{UP} (mW)	Total P _{IO} (mW)
13	EP1S25	780 FineLine BGA	C - commercial	1.5-V	450.00	0.00
14						
15	Import Data	Enter Toggle %	Clear All Values			
16						
17	I_{CC} Standby (mA)					
18	Worst-case	300				
19						
208	The total I_{CCINT} is lower than the power-up I_{CC}; it is advised that the regulator chosen supports the worst case power-up I_{CC} requirement.					
209	TOTAL	I _{CC} (mA)	Power (mW)		Power-Up I_{CC} (mA)	
210	Internal (V _{CCINT})	300.00	450.00		Maximum	1500
211	I/O (V _{CCIO})	0.00	0.00			
212	TOTAL	300.00	450.00			
213						
214	Thermal Analysis					
215	T _j (Degrees C)	T _a (Degrees C)	Required θ_{JA}			
216	85	40	100.00			
217						
218						
219	Thermal Resistance Values for Chosen Device & Package					
220			θ_{JA}			
221	8JC	Still Air	100 LFpM	200 LFpM	400 LFpM	
222	0.25	10.5	8.5	7.1	6	
223						
	◀ ▶ ↻ ↻	Estimator	Version /			

Figure 7–2. Clock Network Section in the Stratix PowerPlay Early Power Estimator

Clock Network				
Global Clock Network	f _{MAX} (MHz)	# Flip-Flops	I _{CCINT} (mA)	P _{INT} (mW)
1	100	984	32.21	48.31
2	20	19	0.43	0.65
3	250	2006	135.97	203.95
4	100	1792	50.09	75.14
5	5	152	0.49	0.74
		Subtotal	219.19	328.78
Regional Clock Network	f _{MAX} (MHz)	# Flip-Flops	I _{CCINT} (mA)	P _{INT} (mW)
1	50	398	6.18	9.27
2	10	2800	5.06	7.59
		Subtotal	11.24	16.86
Fast Regional Clock Network	f _{MAX} (MHz)	# Flip-Flops	I _{CCINT} (mA)	P _{INT} (mW)
1	156	1109	41.85	62.77
		Subtotal	41.85	62.77

Figure 7–3. Logic Elements Section in the Stratix PowerPlay Early Power Estimator

Logic Elements (LEs)						
Average Fan-out						
4.16						
Design Module	f _{MAX} (MHz)	# LEs	# LEs w/Carry	Toggle %	I _{CCINT} (mA)	P _{INT} (mW)
1	250	2500	2000	12.50	86.06	129.08
2	100	1700	1400	12.50	23.53	35.30
3	50	500	400	12.50	3.44	5.16
4	20	511	421	12.50	1.41	2.12
5	10	600	450	12.50	0.82	1.23
6	0	0	0	0.00	0.00	0.00
7	0	0	0	0.00	0.00	0.00
8	0	0	0	0.00	0.00	0.00
9	0	0	0	0.00	0.00	0.00
10	0	0	0	0.00	0.00	0.00
				Subtotal	115.26	172.89

Figure 7–4. RAM Blocks Section in the Stratix PowerPlay Early Power Estimator

RAM Blocks										
M512 Blocks										
Design Module	f _{MAX} (MHz)	# Data Inputs	# Data Outputs	Toggle %	# M512 Blocks Used	Mode	Total I _{CC_READ}	Total I _{CC_WRITE}	I _{CC_WTR} (mA)	P _{WR} (mW)
1	100	9	9	12.50	50	Single-Port	8.47	2.73	11.20	16.80
M4K Blocks										
Design Module	f _{MAX} (MHz)	# Data Inputs	# Data Outputs	Toggle %	# M4K Blocks Used	Mode	Total I _{CC_READ}	Total I _{CC_WRITE}	I _{CC_WTR} (mA)	P _{WR} (mW)
1	100	0	8	12.50	20	ROM	4.63	0.00	4.63	6.94
M-RAM Blocks										
Design Module	f _{MAX} (MHz)	# Data Inputs	# Data Outputs	Toggle %	# M-RAM Blocks Used	Mode	Total I _{CC_READ}	Total I _{CC_WRITE}	I _{CC_WTR} (mA)	P _{WR} (mW)
1	100	48	48	12.50	2	True-Dual-Port	2.55	0.19	2.74	4.11

Figure 7–5. General I/O Power Section in the Stratix PowerPlay Early Power Estimator

General I/O Power											
Design Module	f _{MAX} (MHz)	# Outputs & Bidirectional Pins	Toggle %	Avg. Capacitive Load (pF)	I/O Standard	I/O Data Rate	I _{CCIO} (mA)				
1	156	64	25.00	4	LVDS	SDR	250.07				
2	133	55	12.50	8	3.3-V PCI	SDR	52.65				
3	50	80	12.50	20	3.3-LVTTL/LVCMSO_24	SDR	42.55				
4	66	128	12.50	10	SSTL-16_I	SDR	561.11				

Estimating Power in the Design Cycle

You can estimate power at different stages of your design cycle. 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 feature in the Quartus II software.

Since FPGAs provide the convenience of a shorter design cycle and faster time-to-market, the board design often takes place during the FPGA design cycle, which means the power planning for the device must happen before the FPGA design is complete. If you have not started the FPGA design, or it is not complete, an estimate of the power consumption for the design can be made using the Early Power Estimator spreadsheet. Table 7–1 shows the power estimation flow when using the PowerPlay Early Power Estimator spreadsheet when the FPGA design has not begun.

When the FPGA design is partially complete, the power estimation file generated by the Quartus II software can help to enter values in the Early Power Estimator spreadsheet. After using the Import Data macro to import the power estimation file information into the Early Power Estimator spreadsheet, you can edit the spreadsheet to reflect the device resource estimates for the final design.

Table 7-1. Power Estimation Before FPGA Design has Begun

Steps to Follow		Advantages	Disadvantages
1	Download the PowerPlay Early Power Estimator spreadsheet from the Altera web site	Power estimation can be done before any FPGA design is complete	Accuracy is primarily dependent on user input and estimate of the device resources
2	Manually enter design parameters in the PowerPlay Early Power Estimator spreadsheet		Can be time consuming



For more information about how to generate the power estimation file in the Quartus II software, see “[Quartus II Early Power Estimator File](#)” on page 7–6. For more information about how to use the Import Data macro to import the power estimation file information into the Early Power Estimator spreadsheet, see the *PowerPlay Early Power Estimator User Guide*.

Table 7-2 shows the power estimation flow for the PowerPlay Early Power Estimator spreadsheet when the FPGA design is partially complete.

Table 7-2. Power Estimation When FPGA Design is Partially Complete

Steps to Follow		Advantages	Disadvantages
1	Compile the partial FPGA design in the Quartus II software	Power estimation can be done early in the FPGA design cycle	Accuracy is primarily dependent on user input and estimate of the final design device resources
2	Generate the Power Estimation File in the Quartus II software		
3	Download the PowerPlay Early Power Estimator spreadsheet from the Altera web site		
4	Run the import data macro to automatically populate the PowerPlay Early Power Estimator spreadsheet		
5	Edit the values in the PowerPlay Early Power Estimator spreadsheet to reflect the device resources used in the final design (Optional)		

When the FPGA design is complete, the device power consumption is estimated with the PowerPlay Power Analyzer solution in the Quartus II software. The Quartus II PowerPlay Power Analyzer provides power estimation for Stratix II, Stratix, Stratix GX, Cyclone, HardCopy® Stratix, MAX II, MAX 7000AE, MAX 7000B, and MAX 3000A devices.



For more information about how to use the PowerPlay Power Analyzer feature in the Quartus II software, see the *PowerPlay Power Analyzer* chapter in Volume 3 of the *Quartus II Handbook*.

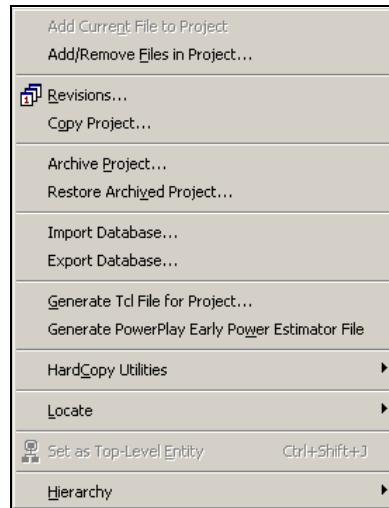
Quartus II Early Power Estimator File

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, then you must estimate the number of device resources used in your design.

If you already have an existing design or a partially completed design, the power estimator file that is generated by the Quartus II software version 4.2 and later can aid in completing the PowerPlay Early Power Estimator spreadsheet.

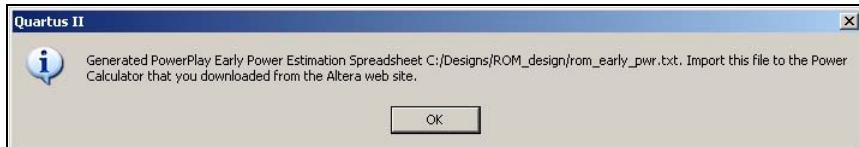
To generate the power estimation file, you must first compile your design in the Quartus II software beginning with version 4.2. After compilation is complete, choose **Generate PowerPlay Early Power Estimator File** (Project menu), which instructs the Quartus II software to write out a power estimator text file. See [Figure 7–6](#).

Figure 7–6. Generate Power Estimation File Option



After the Quartus II software successfully generates the power estimator file, a message is displayed. See [Figure 7–7](#).

Figure 7–7. Generate Power Estimation File Message



The power estimator file is named *<name of Quartus II project>_early_pwr.txt*. **Figure 7–8** is an example of the contents of a power estimation file generated by the Quartus II software version 4.2.

Figure 7–8. Example of Power Estimation File

Power Estimation File for dff_top - Do not edit this line

```
<name=DEVICE value=EP1S25F780C5>

<name=fmax_RC1 value=100>
<name=ff_RC1 value=984>
<name=fmax_LE1 value=100>
<name=tot_LE1 value=1700>
<name=totwcc LE1 value=1400>
<name=fmax_GIO1 value=50>
<name=NumbOB_GIO1 value=80>
<name=avgCLoad_GIO1 value=20>
<name=iostd_GIO1 value=3.3_LVTTL/LVCMOS_24>
<name=iodatarate_GIO1 value=SDR>
```

PowerPlay Early Power Estimator spreadsheets include 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 also transfer the data into the Early Power Estimator spreadsheet manually.

If your existing Quartus II project represents only a portion of your full design, you should enter the additional resources that are used in the final design manually. Therefore, after importing the power estimation file information into the Early Power Estimator spreadsheet, you can edit it to add additional device resources.



For completed designs, see the *PowerPlay Power Analyzer* chapter in Volume 3 of the *Quartus II Handbook*.

Conclusion

The PowerPlay Early Power Estimator spreadsheet is an easy and useful tool to estimate the power consumption for your designs based on typical conditions. The power estimator file generated by the Quartus II software helps to enter data into the Early Power Estimator spreadsheet available on the Altera web site. Board-level and FPGA designers can benefit from the power estimator file generated by the Quartus II software to more accurately estimate power. Power estimation can be refined later in the design cycle using the PowerPlay Power Analyzer in the Quartus II software.



For more information refer to the *PowerPlay Early Power Estimator User Guide*.

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 8, PowerPlay Power Analyzer](#)
- [Chapter 9, Quick Design Debugging Using SignalProbe](#)
- [Chapter 10, Design Debugging Using the SignalTap II Embedded Logic Analyzer](#)
- [Chapter 11, Design Analysis & Engineering Change Management with Chip Editor](#)

Revision History

Chapter 11, *Cadence Incisive Conformal Support* was moved to Section V, in Volume 3 of the *Quartus II Handbook*.

The table below shows the revision history for [Chapters 10 to 11](#).

Chapter(s)	Date / Version	Changes Made
8	Dec. 2004 v1.0	Initial release.
9	Dec. 2004 v2.1	<ul style="list-style-type: none">● Chapter 9 was formerly Chapter 8.● Updates to tables and figures.● New functionality for Quartus II software 4.2.
	June 2004 v2.0	<ul style="list-style-type: none">● Updates to tables, figures.● New functionality for Quartus II software 4.1.
	Feb. 2004 v1.0	Initial release.
10	Dec. 2004 v2.1	<ul style="list-style-type: none">● Chapter 10 was formerly Chapter 9.● Updates to tables and figures.● New functionality for Quartus II software 4.2.
	June 2004 v2.0	<ul style="list-style-type: none">● Updates to tables and figures.● New functionality for Quartus II software 4.1.
	Feb. 2004 v1.0	Initial release.
11	Jan. 2005 v2.2	Change Manager section update
	Dec. 2004 v2.1	<ul style="list-style-type: none">● Chapter 11 was formerly Chapter 10.● New figures added.● Reorganized the chapter and updated information and figures.● New functionality for Quartus II software 4.2.
	June 2004 v2.0	<ul style="list-style-type: none">● Updates to tables and figures.● New functionality for Quartus II software 4.1.
	Feb. 2004 v1.0	Initial release.

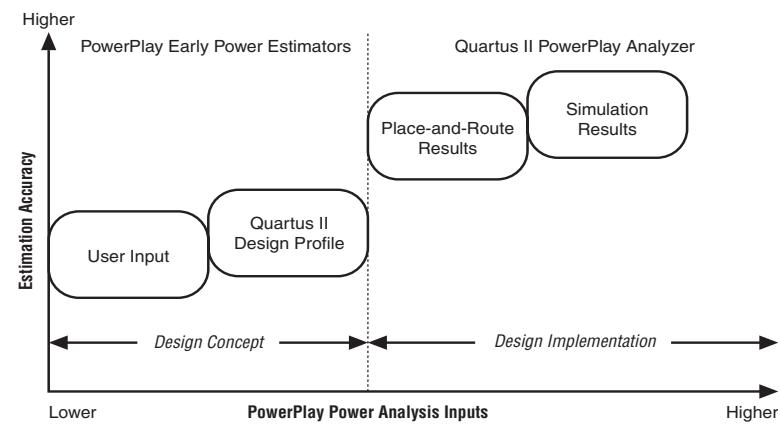
qii53013-1.0

Introduction

This chapter describes the Quartus® II PowerPlay Power Analyzer tool and how to use the tool to accurately estimate device power consumption. The PowerPlay Power Analyzer is run after synthesis and place-and-route are completed. This tool provides the detailed information on the design implementation to enable high-quality power estimates. The PowerPlay Power Analyzer supports power estimation for Stratix® II, Stratix, Stratix GX, Cyclone™, MAX® II, MAX 7000AE, MAX 7000B, and MAX 3000A devices.

The PowerPlay power analysis tools in the Quartus II software give you improved accuracy of power consumption and the ability to estimate power consumption from early design concept through design implementation, as shown in [Figure 8–1](#).

Figure 8–1. PowerPlay Power Analysis



For more information on estimating device power consumption before fitting, see the *PowerPlay Early Power Estimator* chapter in Volume 3 of the *Quartus II Handbook*.

This chapter provides information about how design parameters affect power and step-by-step instructions for using the PowerPlay Power Analyzer to obtain power estimates in a variety of different flows.

Table 8–1 lists the main differences between the PowerPlay Early Power Estimator and the PowerPlay Power Analyzer.

The results of the Power Analyzer must be used only as an estimation of power, not as a specification. The purpose of the estimation is to help establish a guide for power budget. Altera® recommends that the actual power be measured on the board. The total device dynamic current must be verified during device operation, because this estimate is sensitive to many factors, including input vector quantity and quality, and the exact loading conditions dependent on the printed circuit board (PCB) design. Static power consumption must not be based on empirical observation. The values reported by the Power Analyzer or from the databook must be used because the devices tested may not exhibit worst-case behavior.

Table 8–1. Comparison of PowerPlay Early Power Estimator & 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	Quartus II
Accuracy	Medium	Medium to Very High
Data inputs	<ul style="list-style-type: none"> • Resource usage estimates • Clock requirements • Environmental conditions • Toggle Rate 	<ul style="list-style-type: none"> • 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)	<ul style="list-style-type: none"> • Total thermal power dissipation • Thermal static power • Thermal dynamic power • Off-chip power dissipation • Voltage supply currents (2) 	<ul style="list-style-type: none"> • Total thermal power • Thermal static power • Thermal dynamic power • Thermal power by design hierarchy • Thermal power by block type • Off-chip (non-thermal) power dissipation • Voltage supply currents (2)

Notes for Table 8–1:

(1) Early Power Estimator output varies by device family as some features may not be available.

(2) Available only for Stratix II and MAX II device families.

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 most 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 ability of the device to dissipate heat. This can impact your choice of a cooling solution that is 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 the static power consumption of the device, increasing junction temperature and resulting in increased 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

This 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 & Thermal Compound

A heat sink allows more efficient heat transfer from the device to the surrounding 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 (θ_{CA}) of your cooling solution.

Design Resources

The design resource used greatly affects power consumption.

Number, Type & 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 & Type of Logic Elements, Multiplier Elements & RAM Elements

A design with more logic elements, multiplier elements, and RAM elements 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 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 & 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 supports several kinds of global clock networks that span either the entire device or smaller subsets (e.g., 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, since the Quartus II software disables unused branches of a clock automatically.

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 the logic and routing. The Quartus II models assume full rail-to-rail switching. For very high toggle rates, especially on circuit output I/Os, it is possible that the circuit transitions before fully charging downstream capacitance. The result is a slightly conservative prediction of power by the Quartus II PowerPlay Power Analyzer.

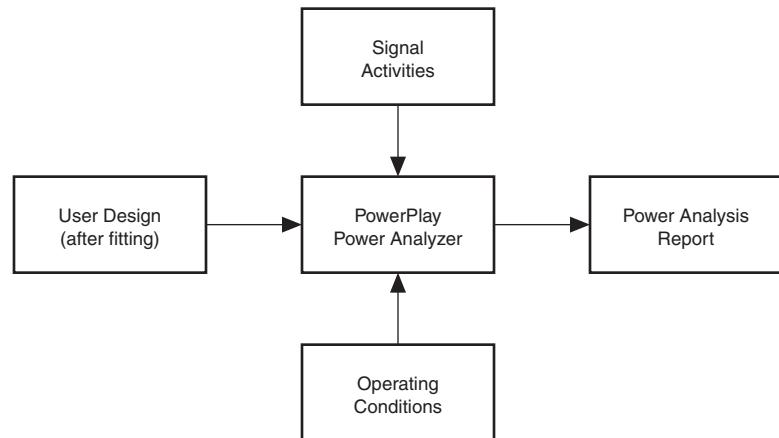
The static power consumed by both routing and logic can sometimes be affected by the static probabilities of their input signals. This effect is due to state-dependent leakage, and is becoming more significant as process geometries shrink. The Quartus II software models this effect on devices at 90 nm (or smaller), if it is deemed important to the power estimate. The static power, due to output I/Os that drive termination resistors, also varies with the static probability of a logic 1 or 0 on the I/O.



To get accurate results from power analysis, the signal activities that are used for the 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 factors affecting power consumption. [Figure 8–2](#) illustrates the high-level Power Analyzer flow.

Figure 8–2. PowerPlay Analyzer High-Level Flow Note (1)**Note for Figure 8–2:**

- (1) Operating condition specifications are available only for the Stratix II and MAX II device families.
-

The PowerPlay 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 output I/O must be specified in the design to obtain accurate I/O power estimates.

Operating Conditions

For the Stratix 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.
- **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

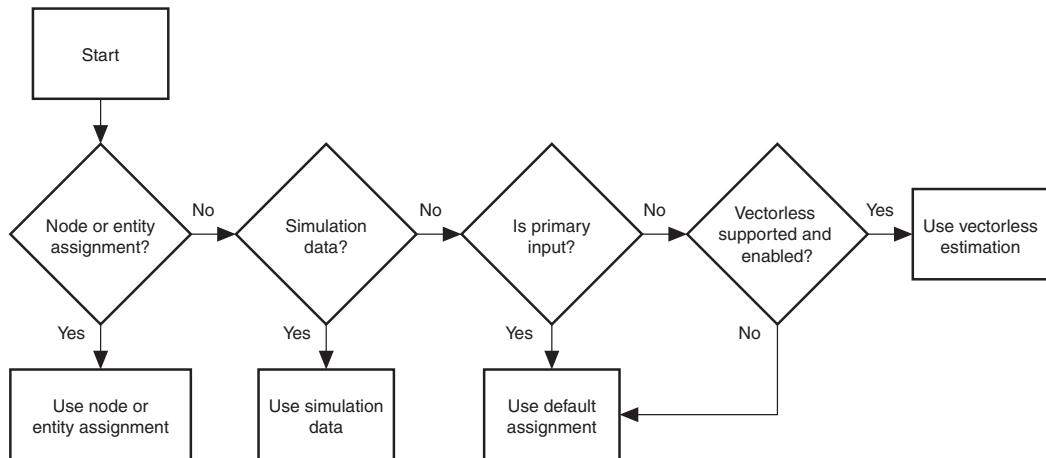
thermal resistance varies with the amount of thermal power that is dissipated. This is because larger heat dissipation increases air convection, thereby reducing thermal resistance. When entering a thermal resistance in such cases, it is important to use the thermal resistance that occurs when the heat flow (Q) is equal to the thermal power generated by the device. Alternatively, the Power Analyzer lets you directly specify a junction temperature. However, this is not the recommended procedure, because you achieve more accurate results by letting the Power Analyzer compute the junction temperature.

Signal Activities Data Sources

The Power Analyzer provides a flexible framework for specifying signal activities. This reflects the critical importance of using representative signal activity data during power analysis. The following data sources can be used:

- Simulation results
- User-entered node, entity, and clock assignments
- User-entered default toggle rate assignment
- Vectorless estimation

The Power Analyzer lets you mix and match the signal activity data sources on a signal-by-signal basis. The priority scheme is shown in [Figure 8–3](#). The data sources are described in the following sections.

Figure 8–3. Signal Activity Data Source Priority Scheme Note (1)**Note to Figure 8–3:**

- (1) Vectorless estimation is available only for the Stratix II and MAX II device families.

Simulation Results

The Power Analyzer can directly read the waveforms generated by a design simulation. The static probability and toggle rate for each signal is calculated from the simulation waveform. Using simulation results is the most accurate way to generate signal activities. Power analysis accuracy is best when simulations are generated using representative input stimuli.

The Power Analyzer reads the results generated by the following simulators:

- Quartus II Simulator
- ModelSim® VHDL, ModelSim Verilog, 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), described in the “[Signal Activities](#)” section. The Quartus II simulator generates an SAF which is then read by the Power Analyzer.

For third-party simulators, use the Quartus II EDA Tool Settings for Simulation to specify **Generate Value Change Dump** file script. These scripts instruct the third-party simulators to generate a Value Change Dump (.vcd) file that encodes the simulated waveforms. The Quartus II Power Analyzer reads the VCD file directly to derive toggle rate and static probability data for each signal.

Third-party EDA simulators, other than those listed above, can be used to 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 VCD file.

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. Essentially, the 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. For more information on how to set the simulation model type for your specific simulator, refer to the online Help.

Glitch filtering in a simulator stops a glitch on one logic element (or other circuit element) output from propagating to downstream circuit elements when the glitch is so fast that the programmable routing filters it out. 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 glitch prone. Hence, 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 third-party simulators, the Power Analyzer flows support two types of glitch filtering. In the first, and recommended level of support, you perform glitch filtering during simulation. To enable this level of glitch filtering in the Quartus II software, turn on the **Glitch filtering** option on the Simulation page of the **EDA Tool Settings** dialog box. The second level of glitch filtering occurs while the Power Analyzer is reading the VCD file generated by the third-party simulator. You enable this by opening the **PowerPlay Power**

Analyzer settings page, selecting a VCD file as input, clicking the **VCD File Options** button, and then selecting the **Perform Glitch Filtering** option in the **VCD File Options** dialog box. Altera recommends that both forms of glitch filtering be used.

The filtering that is performed by the VCD reader is complementary to that performed during simulation, and is often not as effective. While the VCD 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. See [Table 8–2](#) for simulator support.

Table 8–2. Simulator Support for Glitch Filtering	
Simulators	Glitch Filtering Support (Stratix II & MAX II device families only)
Quartus II Simulator	During simulation
3rd Party Simulators	During simulation and reading VCD

Node & 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 “Power Toggle Rate” and “Power Static Probability” assignments.



For more information about how to use assignment editor in the Quartus II software, see 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 for the output pad by appending the string `~result`. For example,

if a design has a bidirectional pin named MYPIN, assignments for the combinational input (to the core) use the name MYPIN, and the assignments for the output pad use the name MYPIN~result.

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 for all other nodes in the design. The default 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 Stratix 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 good accuracy.

Using the PowerPlay Power Analyzer

For all flows in which the PowerPlay Power analyzer is used, the design must first be synthesized and then fit 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 that is 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 PowerPlay Power Analyzer in several ways, but 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. 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 fairly low level of accuracy, because vectorless estimation for registers is not entirely accurate.

Signal Activities From User Defaults Only

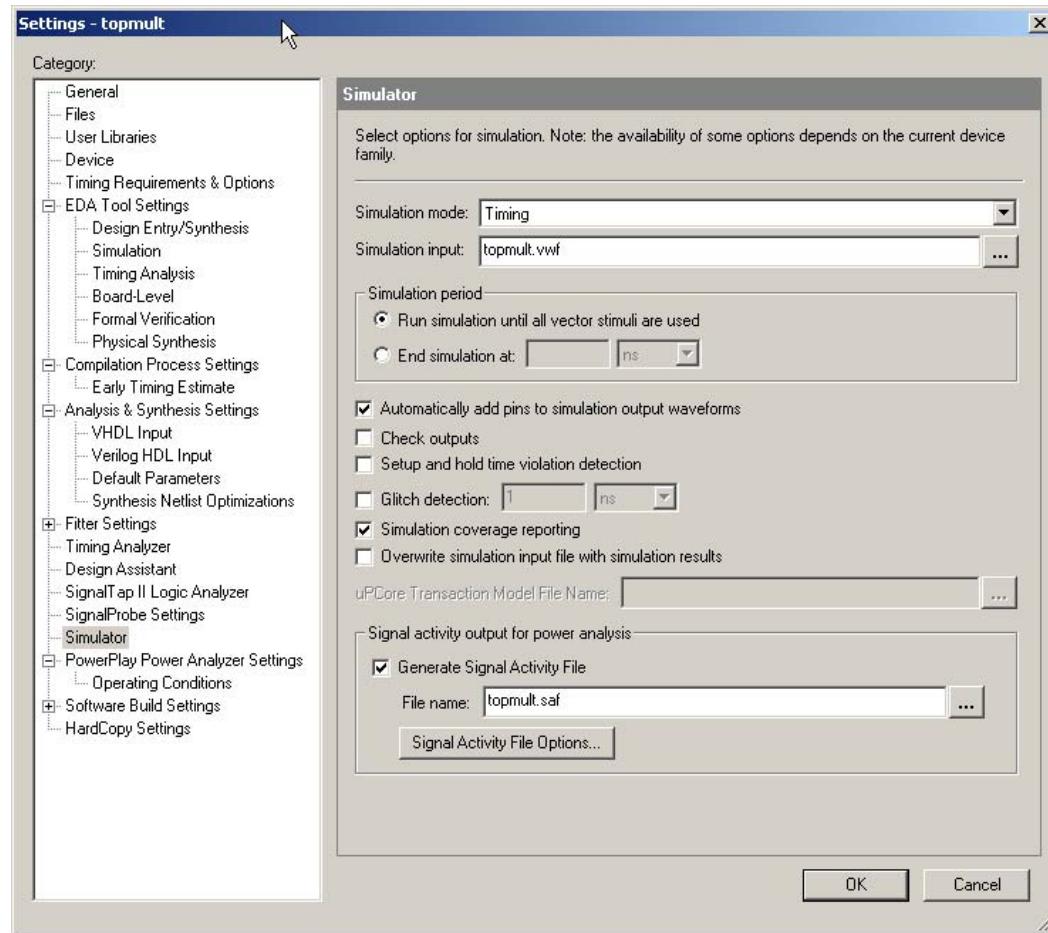
This option provides the lowest degree of accuracy.

Generating a Signal Activity File Using the Quartus II Simulator

While performing a timing or functional simulation using the Quartus II Simulator, you can generate a SAF. The SAF stores 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 as input to the PowerPlay Power Analyzer to estimate power for your design.

To create a SAF for your design, perform the following steps:

1. Choose **Settings** (Assignment menu).
2. In the **Settings** dialog box, under the **Category** list, select **Simulator** (see [Figure 8-4](#)).

Figure 8–4. Simulator Settings

3. In the **Settings** dialog box, select either **Timing** or **Functional** in the **Simulation mode** list. Refer to the “[Common Analysis Flows](#)” section for a description of the difference in accuracy between the two types of simulation modes.
4. Turn on **Generate Signal Activity File** and enter the file name for the SAF file in the **File name** field.
5. (Optional) Click **Signal Activity File Options** to open the Signal Activity File Options window (see [Figure 8–5](#)).

Figure 8–5. Signal Activity File Options

6. (Optional) In the **Signal Activity File Options** dialog box, 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 test benches. 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.

7. (Optional) In the **Signal Activity File Options** dialog box, turn on **Perform glitch filtering**. For more information on glitch filtering, refer to the “[Glitch Filtering](#)” section.
8. After the simulation is complete, a SAF is generated with the specified filename and stored in the main project directory.



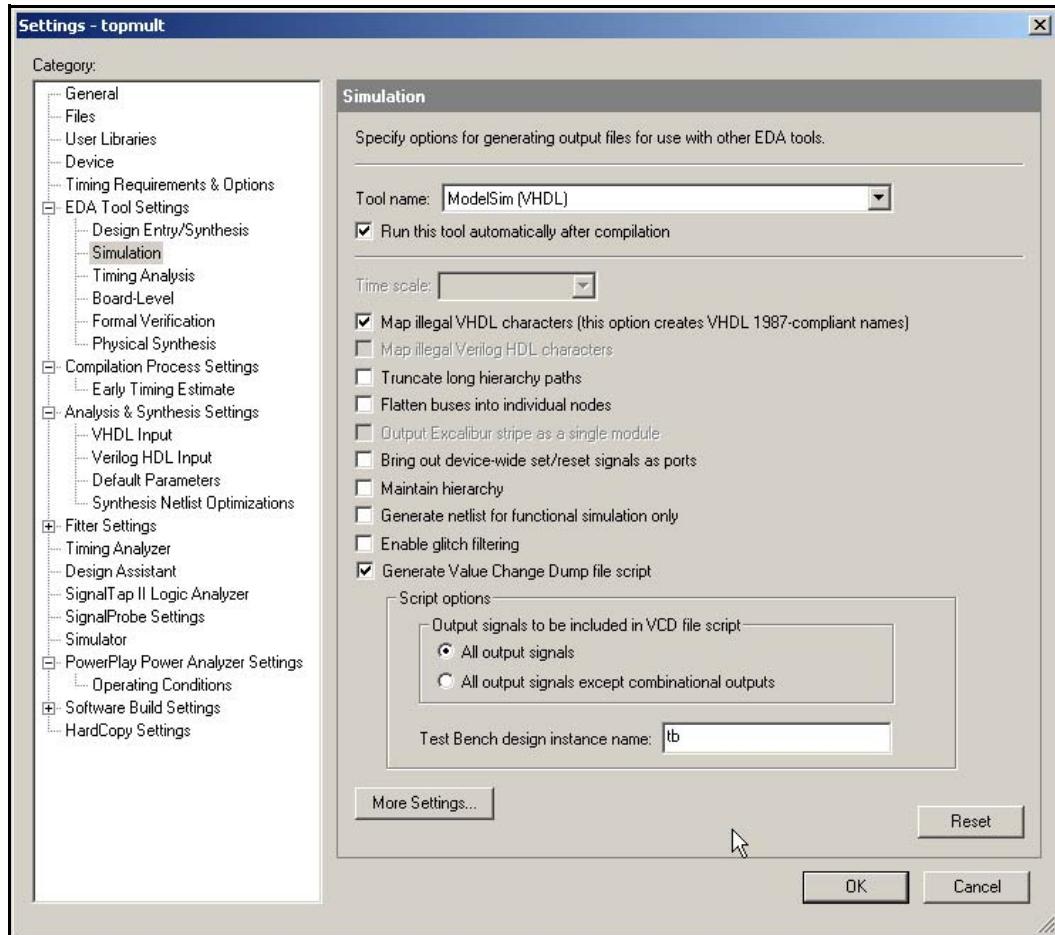
For more information about how to perform simulations in the Quartus II software, see the Quartus II Help.

Generating a Value Change Dump File Using a Third-Party Simulator

You can use other EDA simulation tools, such as the Model TechnologyTM ModelSim software, to perform a simulation and create a VCD file. You can use the VCD 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 on the supported third-party simulators, refer to the “[Simulation Results](#)” section.

To create a VCD file for your design, perform the following steps:

1. Choose **EDA tool settings** (Assignments menu).
2. In the EDA tools **Settings** dialog box, under the **Category** list, select **Simulation** (see [Figure 8–6](#)).

Figure 8–6. EDA Tool Simulation Setting Window

3. In the **Simulation** settings page, choose the appropriate EDA simulation tool in the **Tool name** list.
4. Turn on **Generate Value Change Dump file scripts** and select which output signals should be output to the VCD file in the **Script options** section. With **All output signals** selected, the generated script instructs the third-party simulator to write all connected output signals to the VCD file. With **All output signals except combinational 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 VCD 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). For families that support vectorless estimation, Altera recommends that you use **All output signals except combinational outputs** because vectorless estimation is fairly accurate at calculating combinational output signal activities.

5. Specify a name in the **Test Bench design instance name** box.
6. (Optional) Turn on **Enable glitch filtering**. By turning on this option, glitch filtering logic is 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 Dump file scripts. For more information on glitch filtering, refer to the “[Glitch Filtering](#)” section.
7. Compile your design in the Quartus II software and generate the necessary EDA netlist and script that tells the third-party simulator to generate a VCD file.
8. 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.

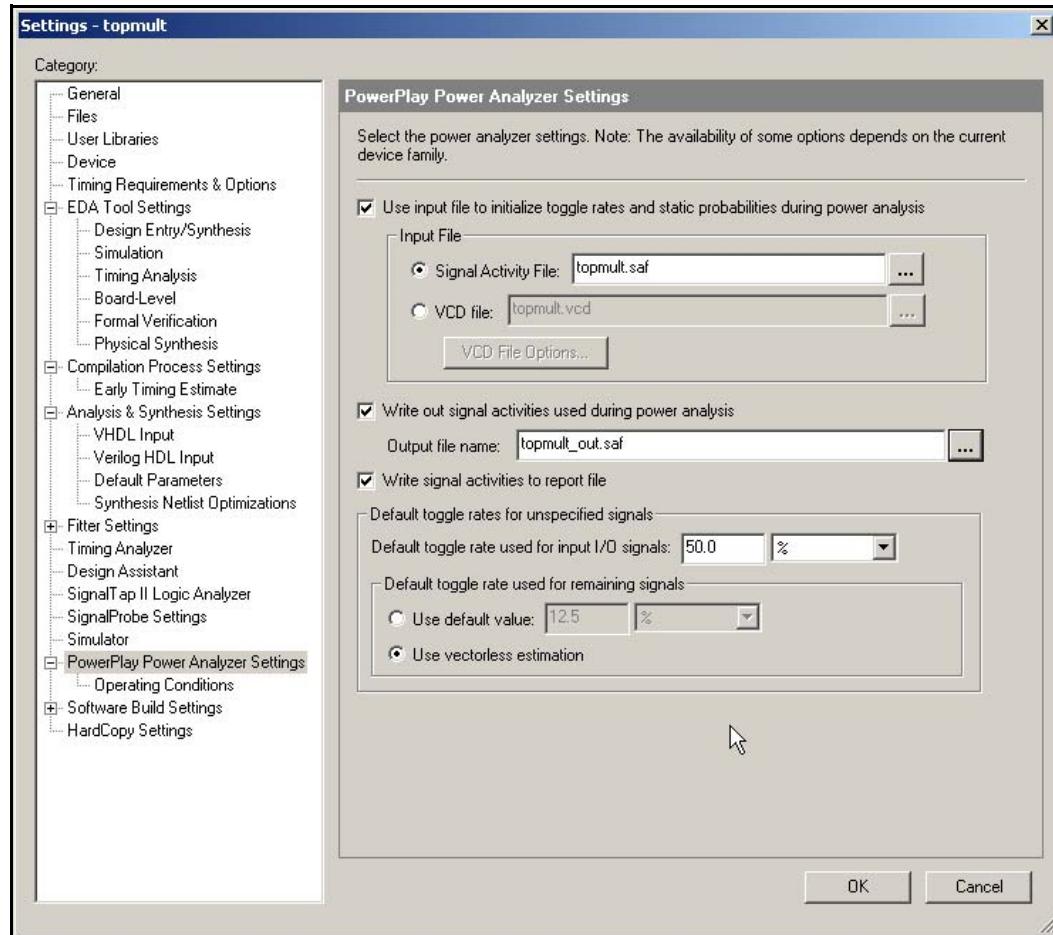


For more information on how to call the VCD generation script in the respective third-party EDA simulation tools, please refer to the Quartus II Help. For more information about how to perform simulations in other EDA simulation tools, see the relevant documentation for that tool.

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. Choose **Settings** (Assignment menu).
2. In the **Settings** dialog box, under the **Category** list, select **PowerPlay Power Analyzer Settings** (see [Figure 8-7](#)).

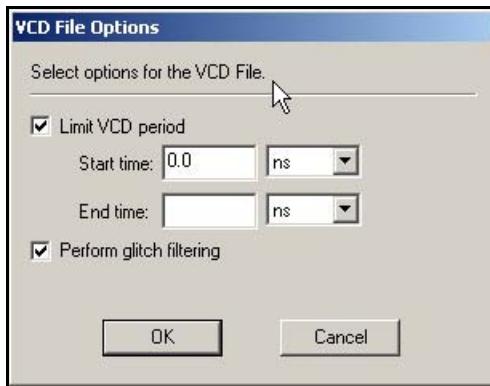
Figure 8–7. PowerPlay Power Analyzer Settings

3. (Optional) If you want to use either a SAF or VCD file as input to the PowerPlay Power Analyzer, turn on **Use input file to initialize toggle rates and static probabilities during power analysis**.
 - a. If you are using a SAF as input, select the correct file from the **Signal Activity File** list under **Input File**. The SAF can be generated by either the Quartus II Simulator or PowerPlay Power Analyzer.

- b. If you are using a VCD file as input, select the correct file from the VCD file list. You can specify multiple VCD filenames using a comma (,) as a separator.

(Optional) Click **VCD File Options** to open the VCD File Options window (see [Figure 8–8](#)).

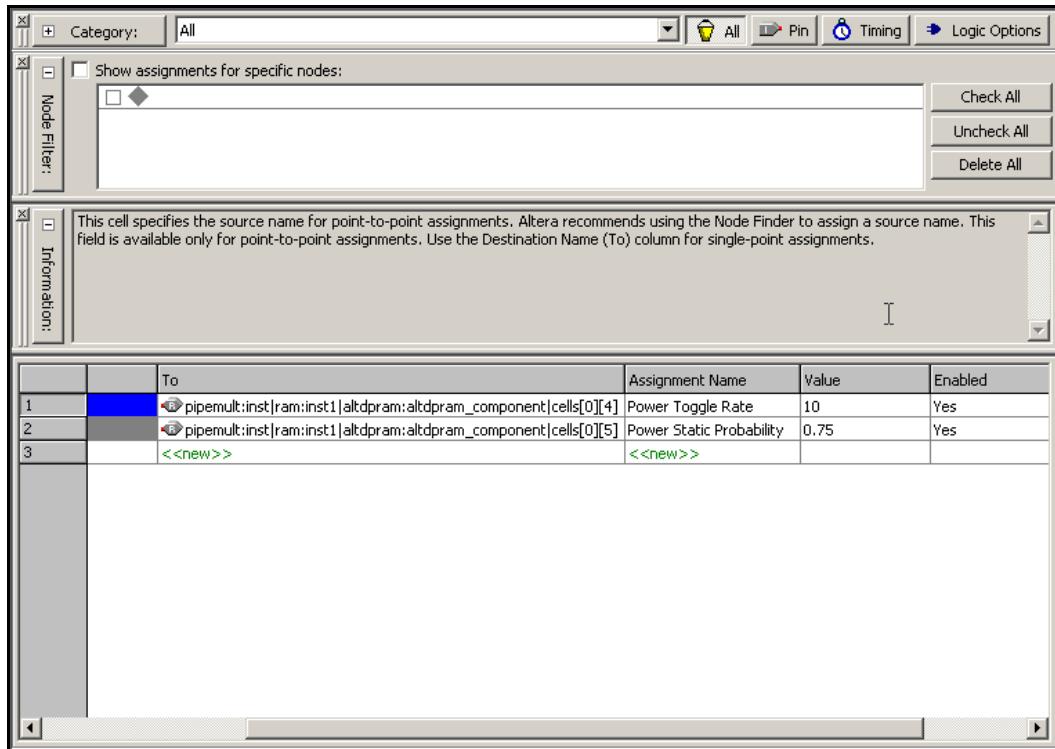
Figure 8–8. VCD File Options



(Optional) In the VCD File Options window, specify the simulation period to use when calculating the signal activities. For more information, refer to step 6 in the “[Generating a Signal Activity File Using the Quartus II Simulator](#)” section.

(Optional) Turn on **Perform glitch filtering**. This is recommended if glitch filtering was not performed when generating the VCD file. For more information, refer to the “[Glitch Filtering](#)” section.

4. (Optional) Turn on **Write out signal activities used during power analysis** and 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 VCD files as input into the PowerPlay Power Analyzer, because it reduces the run time of any subsequent power estimation. The generated SAF can be used as input instead of the original VCD files.
5. (Optional) You can also use the Assignment Editor to enter the Power Toggle Rate and Power Static Probability for a node or entity in your design.

Figure 8–9. Assignment Editor Notes (1), (2)**Notes:**

- (1) The assignments made with the Assignment Editor override the values already existing in SAF or VCD files.
- (2) You can also use Tcl script commands to make these assignments.

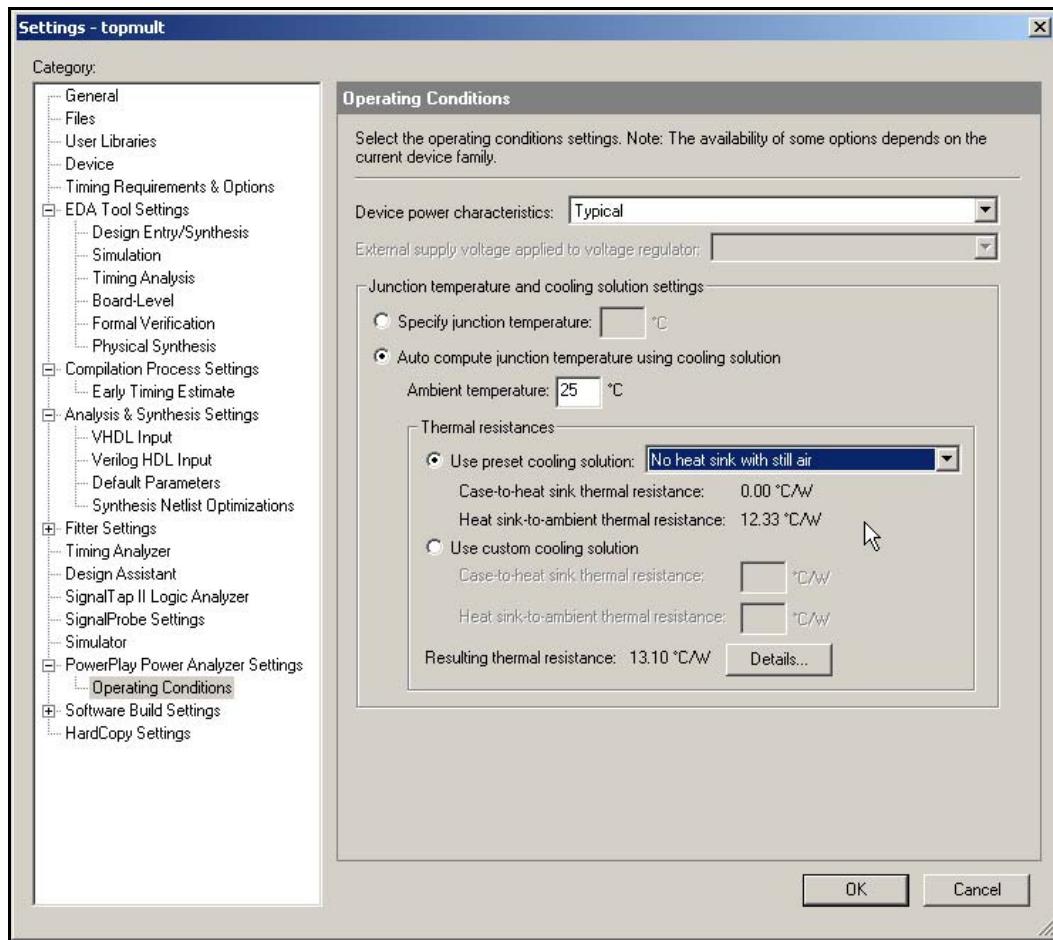


For more information about how to use assignment editor in Quartus II software, see the *Assignment Editor* chapter in Volume 2 of the *Quartus II Handbook*. For information on scripting, see the *Scripting* chapter in Volume 2 of the *Quartus II Handbook*.

6. (Optional) Turn on **Write signal activities to report file**.
7. 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 can not be changed.

8. For Stratix II or MAX II device families, select either **Use default value** or **Use vectorless estimation**. 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 the “Vectorless Estimation” and “Default Toggle Rate Assignment” sections.
9. In the **Settings** dialog box, select **Operating Conditions** under **PowerPlay Power Analyzer Settings**. This option is available only for the Stratix II and MAX II device families (see [Figure 8–10](#)).

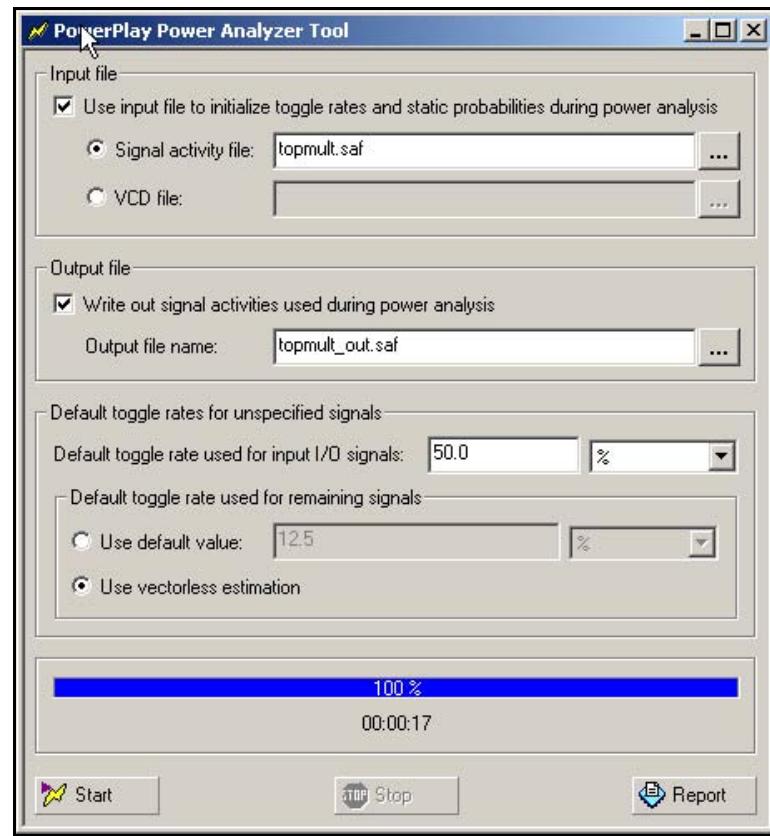
Figure 8–10. Operating Conditions



10. Select **Typical** or **Maximum** from the **Device power characteristics** list. The default is **Typical**.
11. Specify the junction temperature and cooling solution settings. You can select **Specify junction temperature** or **Auto compute junction temperature using cooling solution**.

For more information on how to use the operating condition settings, refer to the “[Operating Conditions](#)” section.
12. Click **OK** to close the **Settings** dialog box.
13. Choose **PowerPlay Power Analyzer** (Tools menu) to open the PowerPlay Power Analyzer Tool window (see [Figure 8–11](#)).

Figure 8–11. PowerPlay Power Analyzer Tool



14. Click **Start** to run the PowerPlay Power Analyzer. Be sure that all the settings are correct.



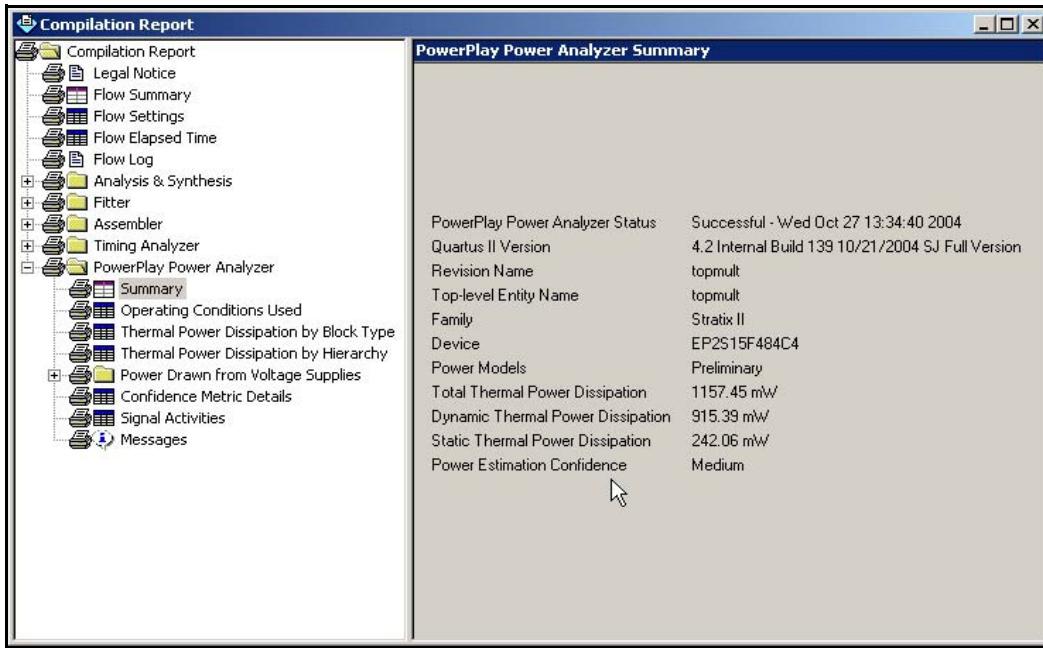
You can also make changes to some of your settings in this window.

15. After the PowerPlay Power Analyzer successfully runs, a message appears (see [Figure 8–12](#)).

Figure 8–12. PowerPlay Power Analyzer Message



-
16. In the PowerPlay Power Analyzer Tool window, click **Report** to open the PowerPlay Power Analyzer Summary window. You can also view the summary in the **PowerPlay Power Analyzer Summary** section of the **Compilation Report** dialog box (see [Figure 8–13](#)).

Figure 8–13. PowerPlay Power Analyzer Summary

PowerPlay Power Analyzer Compilation Report

The PowerPlay Power Analyzer section of Compilation Report is divided into the following sections.

Summary

This section of the report shows the estimated total thermal power consumption of your design. This includes both dynamic and static power consumption. The report also includes a confidence metric that reflects the overall quality of the data sources for the signal activities.

Operating Conditions Used

This section shows device characteristics, voltages, 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 Stratix II and MAX II device families.

Thermal Power Dissipation by Block Type (Device Resource Type)

This section 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. Thermal power is the power dissipated as heat from the FPGA device.

Thermal Power Dissipation by Hierarchy

This section 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 is very useful in locating problem modules in your design.

Power Drawn From Voltage Supplies Supports Power Supply Planning

This section lists the power that was drawn from each voltage supply. The V_{CCIO} voltage supply is further categorized by I/O bank and by voltage. This page is created only for Stratix II and MAX II device families.

Confidence Metric Details

This section provides information about the quality of the signal activity data sources. This section reports the number of each type of data sources for each type of signal (Pin, Registered, or Combinational). For example, the Simulation data source and the Node or entity assignment data sources are considered inherently more trustworthy than vectorless estimation or user-specified default assignments. The section provides enough details so you can decide on your own confidence summary.

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 (one of Pin, Registered, and 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 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 with a Tcl script or at a command prompt.



For more information about Tcl scripting, see the *Tcl Scripting* chapter in Volume 2 of the *Quartus II Handbook*. For more information about command-line scripting, see the *Command-Line Scripting* chapter in Volume 2 of the *Quartus II Handbook*.

For detailed information about scripting command options, type the following at a system command prompt:

```
quartus_sh --qhelp ↵
```

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 ↵
```

Example of using the **quartus_pow** executable with project **sample.qpf**:

1. To instruct the PowerPlay Power Analyzer to use a SAF file as input (**sample.saf**), type the following at a system command prompt:

```
quartus_pow sample --input_saf=sample.saf ↵
```

2. To instruct the PowerPlay Power Analyzer to use two VCD files as input (**sample1.vcd** and **sample2.vcd**), perform glitch filtering on the VCD files, and to 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 ↵
```

3. 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](#)” section.

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 power budget.

qii53008-2.1

Introduction

Hardware verification can be a lengthy and expensive process. The SignalProbe™ incremental routing feature can help 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 the debugging of a design. The SignalProbe feature enables efficient design verification by allowing you to quickly route 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.

The SignalProbe feature supports the MAX® II, Stratix® II, Stratix, Stratix GX, Cyclone™, APEX™ II, APEX 20KE, APEX 20KC, APEX 20K, and Excalibur™ device families.

 You can accomplish the same functionality with the Chip Editor as you can with SignalProbe.



For more information about using the Chip Editor to perform SignalProbe functionality, see the *Design Analysis and Engineering Change Management with Chip Editor* chapter in Volume 3 of the *Quartus® II Handbook*.

Using SignalProbe

You can use the SignalProbe compilation to incrementally route internal signals to output pins. This process completes in a fraction of the time required by a full design recompilation. The incremental routing does not affect source behavior or design operation.

Follow the steps below to use the SignalProbe incremental routing feature:

1. Reserve SignalProbe pins.
2. Assign a SignalProbe source to each SignalProbe pin.
3. Assign an I/O standard to the SignalProbe pins.
4. Add registers for pipelining of signals, if necessary.

5. Perform a SignalProbe compilation.
6. Analyze the results of the SignalProbe compilation.

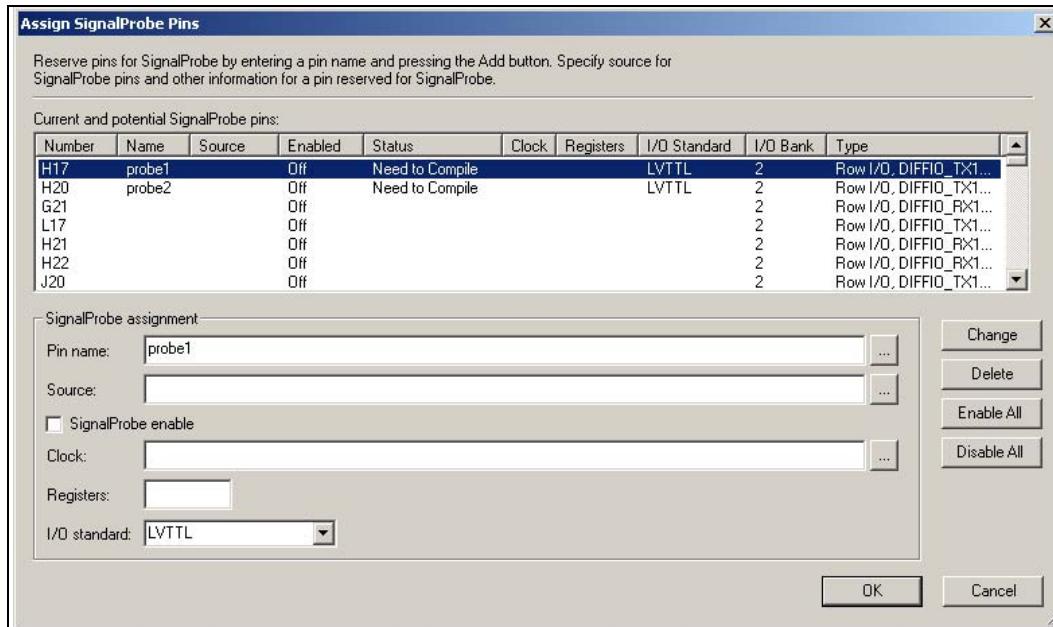
Reserving SignalProbe pins

You can reserve your SignalProbe pins before or after a compilation. Altera recommends that you reserve your SignalProbe pin prior to compilation to ensure that pins are available for SignalProbe use and are not used by another unassigned user I/O pin.

You may need only a few SignalProbe pins, since you can easily reassign different sources to your SignalProbe pins.

To reserve an unused I/O pin as a SignalProbe pin, perform the following steps:

1. Click **Assign SignalProbe Pins** on the **SignalProbe Settings** page of the **Settings** dialog box (Assignment menu). See [Figure 9–1](#).
2. Select a pin **Number** from the **Current and potential SignalProbe pins** list.
3. Type your SignalProbe pin name into the **Pin name** box.
4. Click **Add** for a new SignalProbe pin.
5. Click **OK**.

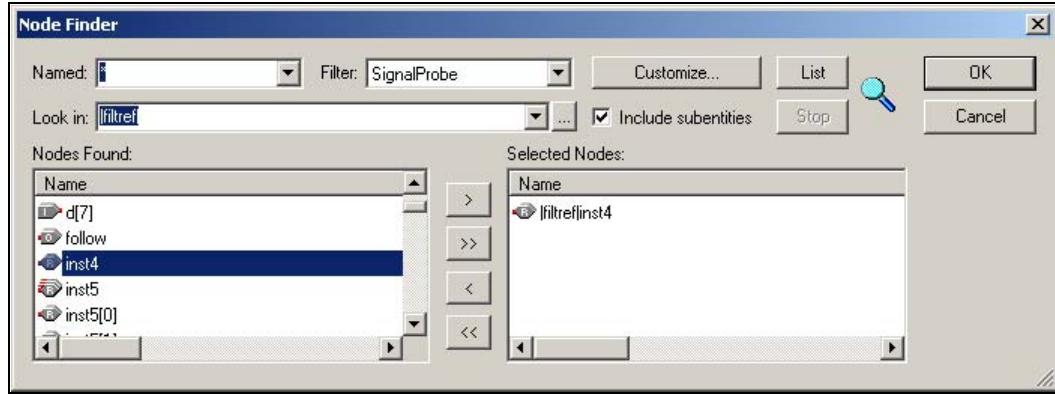
Figure 9–1. Reserving a Pin for SignalProbe in the Assign SignalProbe Pins Dialog Box

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. Click **Assign SignalProbe Pins** on the **SignalProbe Settings** page of the **Settings** dialog box (Assignments menu).
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 (...)**, select a **SignalProbe source** and click **OK**.

The **Node Finder** dialog box appears with the SignalProbe filter selected (see [Figure 9–2](#)). Click **List** to view all the available SignalProbe sources. If you cannot find a specific node with the SignalProbe filter, then the node has been either removed by the Quartus II software during optimization or placed somewhere in the device where there are no possible routes to a pin.

Figure 9–2. Available SignalProbe Sources in the Node Finder

4. 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 modified, the SignalProbe pin is automatically enabled. To disable a SignalProbe pin, turn off **SignalProbe enable**.

5. Click **OK**

Assigning I/O Standards

The I/O standard of each SignalProbe pin must be compatible with the I/O bank the pin is in.

You can use the following two methods to assign I/O standards for your SignalProbe pins:

- Click **Assign SignalProbe Pins** on the **SignalProbe Settings** page of the **Settings** dialog box (Assignments menu), select your SignalProbe output and select an I/O standard from the **I/O Standard** list.
- or*
- Choose **Assignment Editor** (Assignments menu), select **I/O Standard** in the **Category** list, type the SignalProbe pin name in the **To** column and select the I/O standard in the **I/O Standard** column of the spreadsheet.

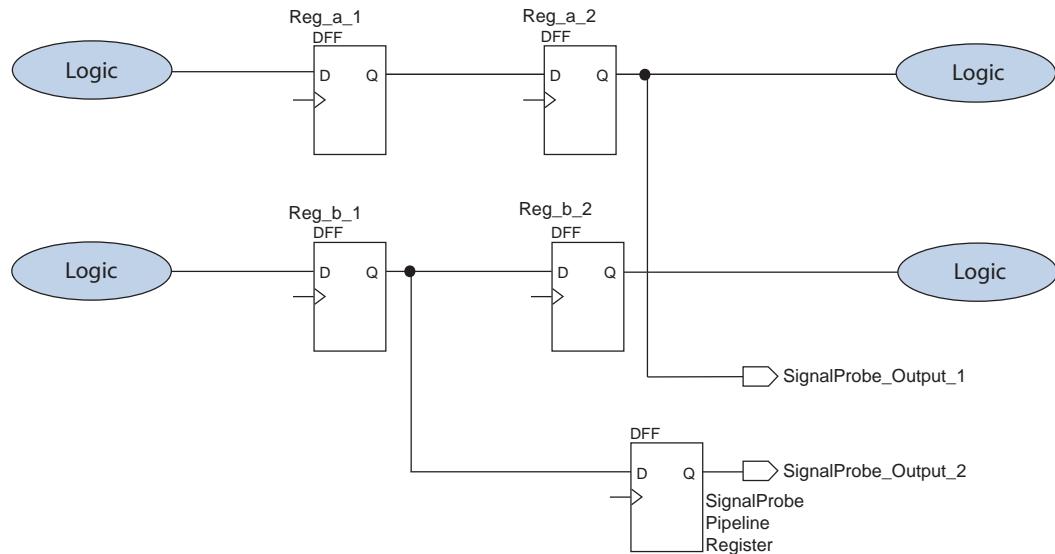
Adding Registers for Pipelining

You can specify the number of registers to be placed between a SignalProbe source and a SignalProbe pin to synchronize the data with a clock and to control the latency. The SignalProbe incremental routing feature automatically inserts the number of registers specified into the SignalProbe path.

For example, you can add a single register between the SignalProbe source and the SignalProbe output pin to reduce the propagation time (t_{CO}). See [Figure 9–3](#). Or you can add multiple registers to your SignalProbe output pins to synchronize the data with other output pins in your design.



When you add one register to a SignalProbe pin, the SignalProbe compilation always attempts to place the register into the I/O element. If it is unable to place the register into the I/O element, it places the register as close to the SignalProbe pin as possible to reduce the clock to output delays (t_{CO}).

Figure 9–3. Synchronizing SignalProbe Outputs with a SignalProbe Register

You can add registers to your SignalProbe pin by performing the following steps:

1. Click **Assign SignalProbe Pins** on the **SignalProbe Settings** page of the **Settings** dialog box (Assignments menu).
2. In the **Current and potential SignalProbe pins** list, select the SignalProbe output pin you want to register.
3. Under **SignalProbe assignment**, type a clock name in the **Clock** box or browse to a clock.
4. Under **SignalProbe assignment**, type the number of registers to pipeline your SignalProbe source in the **Registers** box.



Altera® strongly recommends using global clock signals to clock the added registers.

The MAX II, Stratix II, Stratix, Stratix GX, and Cyclone support adding registers to a SignalProbe pin.

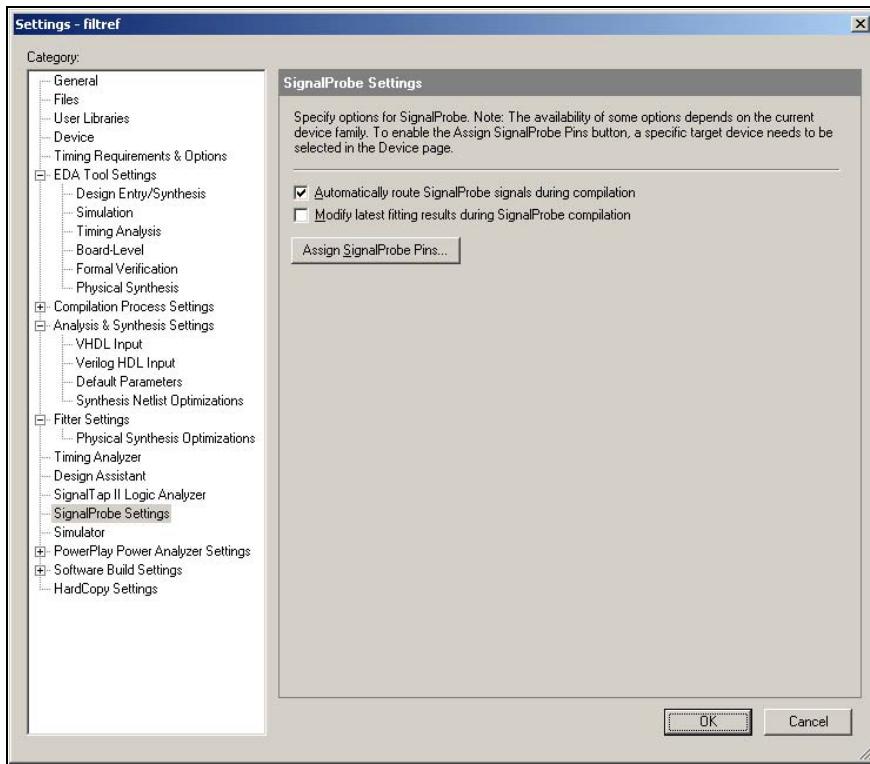
Performing a SignalProbe Compilation

You can start a SignalProbe compilation manually or automatically after a full compilation. A SignalProbe compilation performs the following steps:

1. Validate SignalProbe pins.
2. Validate your specified SignalProbe sources.
3. If applicable, add registers into SignalProbe paths.
4. Attempt to route from SignalProbe sources, through registers, to SignalProbe pins.

To make the SignalProbe compilation run automatically after a full compile, turn on **Automatically route SignalProbe sources during compilation** in the **SignalProbe Settings** page in the **Settings** dialog box (Assignments menu), (see [Figure 9–4](#)).

Figure 9–4. SignalProbe Settings Page in the Settings Dialog Box



To run a SignalProbe compilation manually after a full compilation, choose **Start SignalProbe Compilation** (Processing -> Start menu).



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 on and off the **SignalProbe enable** option in the **Assign SignalProbe Pins** dialog box. You can also enable or disable all SignalProbe pins by clicking **Enable All** and **Disable All** respectively in the **Assign SignalProbe Pins** dialog box.

Running SignalProbe with Smart Compilation

Smart compilation 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, turn on **Use Smart compilation** in the **Compilation Process** page in the **Settings** dialog box (Assignments menu).

If you run a SignalProbe compilation with smart compilation on, and there are changes to a design file or settings related to the Analysis and Synthesis or Fitter modules, then you get the following message:

Error: Can't perform SignalProbe compilation because design requires a full compilation.



Altera recommends turning on smart compilation which allows you to work with the latest settings and design files.

Analyzing SignalProbe Routing Failures

If the SignalProbe compilation starts and then fails, one of the following could be the reason:

- Route unavailable—The SignalProbe compilation failed to find a route from the SignalProbe source to the SignalProbe pin because of routing congestion
- Invalid or non-existent 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

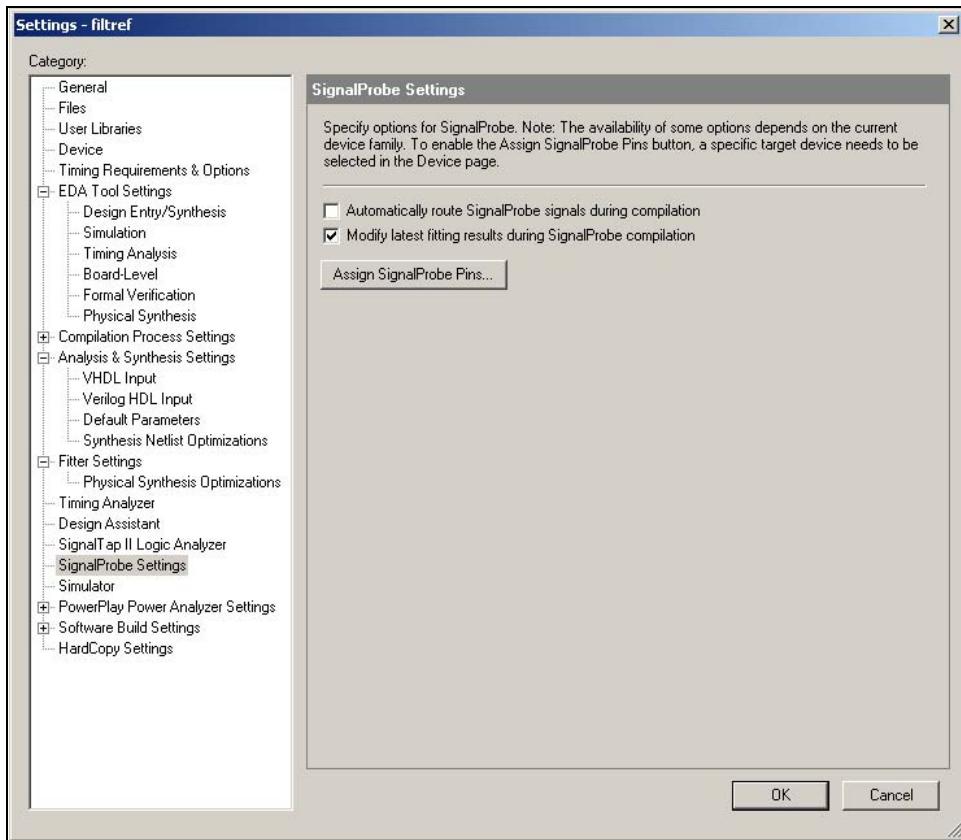
Routing failures can occur if the SignalProbe pin's I/O standard conflicts with other I/O standards in the same I/O Bank.

If routing congestion is preventing a successful SignalProbe compilation, you can turn on **Modify latest fitting results during SignalProbe compilation** in the **SignalProbe Settings** page in the **Settings** dialog box (Assignments menu) to allow the compiler to modify the routing to the specified SignalProbe source (see [Figure 9–5](#)). This setting allows the Fitter to modify the existing routing channels used by your design.



Turning on **Modify latest fitting results during SignalProbe compilation** may change the performance of your design.

Figure 9–5. SignalProbe Settings Page in the Settings Dialog Box



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 (see [Table 9–1](#)) of each SignalProbe pin is displayed in the SignalProbe Fitting Result page in the Fitter section of the compilation report (see [Figure 9–6](#)). The SignalProbe Fitting Results page also shows the status of each SignalProbe pin (see [Table 9–1](#)).

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 (see [Figure 9–7](#)).

 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 9–1. 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 9–6. SignalProbe Fitting Results Page in the Compilation Report Window

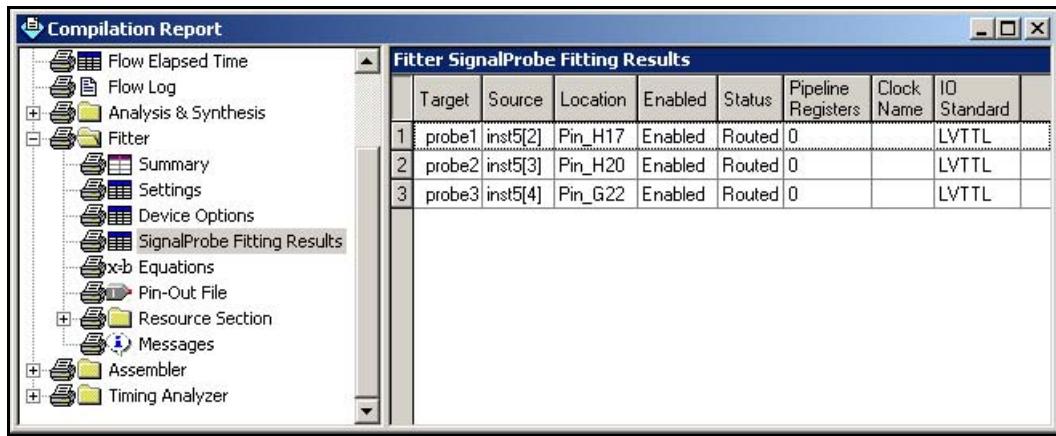
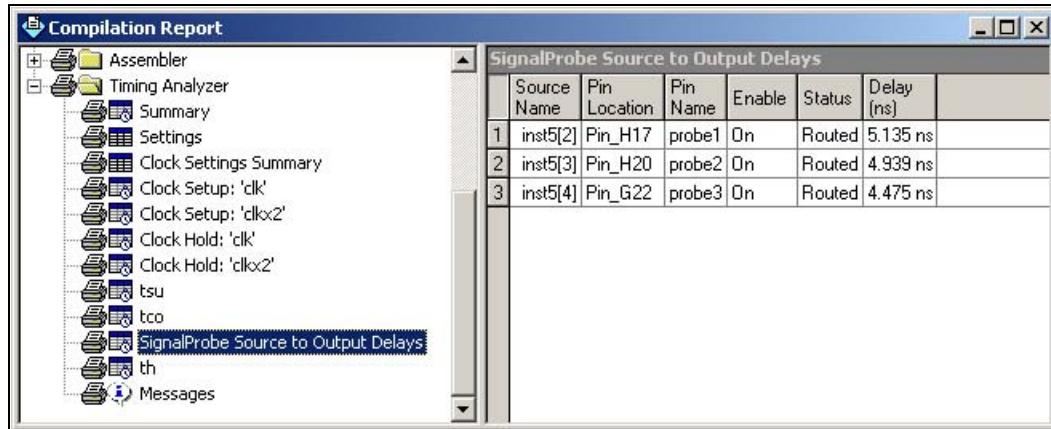


Figure 9–7. SignalProbe Source to Output Delays Page in the Compilation Report Window



Scripting Support

You can run procedures and make settings described in this chapter into a Tcl script. 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 `quartus_sh --qhelp` at a system command prompt to run the Quartus II Command-Line and Tcl API Help browser.



For more information on Quartus II scripting support, including examples, refer to *Command-Line Scripting* and *Tcl Scripting* chapters in Volume 2 of the *Quartus II Handbook*.

Reserving SignalProbe Pins

Use the following Tcl commands to reserve a SignalProbe pin. For more information about reserving SignalProbe pins, see “[Reserving SignalProbe pins](#)” on page 9–2.

```
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`.

Adding SignalProbe Sources

Use the following Tcl commands to add SignalProbe sources. For more information about adding SignalProbe sources, see “[Adding SignalProbe Sources](#)” on page 9–3. 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 enables the SignalProbe routing. You can disable individual SignalProbe pins by specifying OFF instead of ON.

```
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. For more information about assigning I/O standards, see “[Assigning I/O Standards](#)” on page 9–5.

```
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. For more information about adding registers for pipelining, see “[Adding Registers for Pipelining](#)” on page 9–5.

```
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 cause SignalProbe to run automatically after a full compile. For more information about running SignalProbe automatically, see “[Performing a SignalProbe Compilation](#)” on page 9–7.

```
set_global_assignment -name SIGNALPROBE_DURING_NORMAL_COMPILATION ON
```

Run SignalProbe Manually

You can run SignalProbe manually with a Tcl command or with a command run at a command prompt. For more information about running SignalProbe manually, see “[Performing a SignalProbe Compilation](#)” on page 9–7.

Tcl command: execute_flow -signalprobe

The execute_flow command is in the flow package.

Command prompt: quartus_fit <project name> --signalprobe ↵

Enable or Disable All SignalProbe Routing

Use this Tcl code to enable or disable all SignalProbe routing. For more information about enabling or disabling SignalProbe routing, see [page 9–7](#). In the set_instance_assignment command, specify ON to enable all SignalProbe routing or OFF to disable all SignalProbe routing.

```
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> \
}
```

Running SignalProbe with Smart Compilation

Use the following Tcl command to turn on **Smart Compilation**. For more information, see “[Running SignalProbe with Smart Compilation](#)” on [page 9–8](#).

```
set_global_assignment -name SPEED_DISK_USAGE_TRADEOFF SMART
```

Allow SignalProbe to Modify Fitting Results

Use the following Tcl command to turn on **Modify latest fitting results**. For more information, see “[Analyzing SignalProbe Routing Failures](#)” on page 9–8.

```
set_global_assignment -name SIGNALPROBE_ALLOW_OVERUSE ON
```

Conclusion

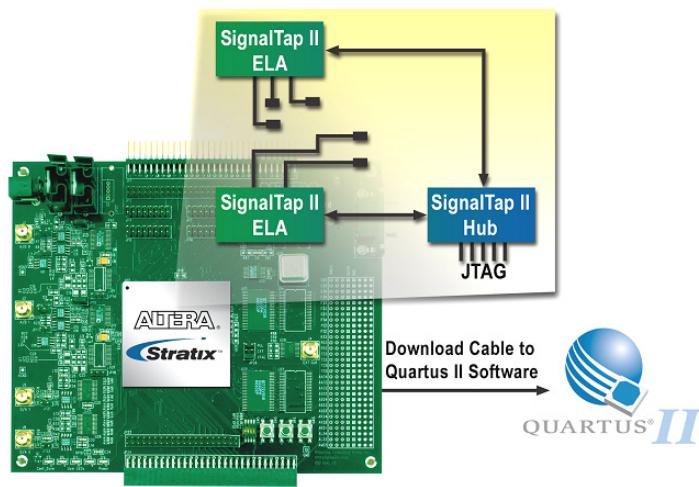
Using the SignalProbe incremental routing feature can significantly reduce the time required for a full recompilation. You can use the SignalProbe incremental routing feature to get quick access to internal design signals to perform system-level debugging.

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, FPGA packages, and printed circuit board (PCB) electrical noise are all contributing factors in making design debugging the most difficult process of the design cycle. You can easily spend more than 50% of your design cycle time on debugging and verifying your design. To help you with the process of design debugging, Altera provides a solution that allows you to examine the behavior of internal signals, without using extra I/O pins, while your design is running at full speed on your FPGA.

The SignalTap® II Logic Analyzer is non-intrusive, scalable, easy to use, and free with your Quartus® II software subscription. This logic analyzer helps you debug your FPGA design by allowing you to probe the state of the internal signals in your design without the use of any extra probes. The SignalTap II embedded logic analyzer does not require any external probes or changes to your design files to capture a design's state of internal nodes or I/O pins.

The SignalTap II Logic Analyzer is a second-generation system-level debugging tool that captures and displays real-time signal behavior in a system on a programmable chip (SOPC). The SignalTap II Logic Analyzer supports the highest number of channels, sample depth, and clock speeds of any embedded logic analyzer in the programmable logic market. You can define custom trigger-condition logic to provide greater accuracy and enhances the ability to isolate problems. [Figure 10–1](#) shows the components of the SignalTap II embedded logic analyzer.

Figure 10–1. SignalTap II Logic Analyzer Block Diagram

The following components are required to perform logic analysis with the SignalTap II embedded logic analyzer:

- Quartus II design software
- Download cable
- Device Under Test

Data is captured on the rising-edge of the sample clock, stored in the device's memory blocks, and streamed out to the Quartus II software waveform display using a JTAG communication cable such as an EthernetBlaster or USB Blaster™. **Table 10–1** summarizes some of the features and benefits of the SignalTap II embedded logic analyzer.

Table 10–1. SignalTap II Features & Benefits (Part 1 of 2)

Feature	Benefit
Multiple logic analyzers in a single device	Allows you to capture data from multiple clock domains in your design
Multiple logic analyzers in multiple devices in a single JTAG chain	Allows you to simultaneously capture data from multiple devices in a JTAG chain
Up to 10 basic or advanced trigger levels for each analyzer	Allows for more complex data capture commands to be given to the logic analyzer, providing greater accuracy and problem isolation
Flexible buffer acquisition modes	Each trigger can be set up to sample at different ranges relative to the triggering event, in circular or segmented modes, which allows more accurate data collection

Table 10–1. SignalTap II Features & Benefits (Part 2 of 2)

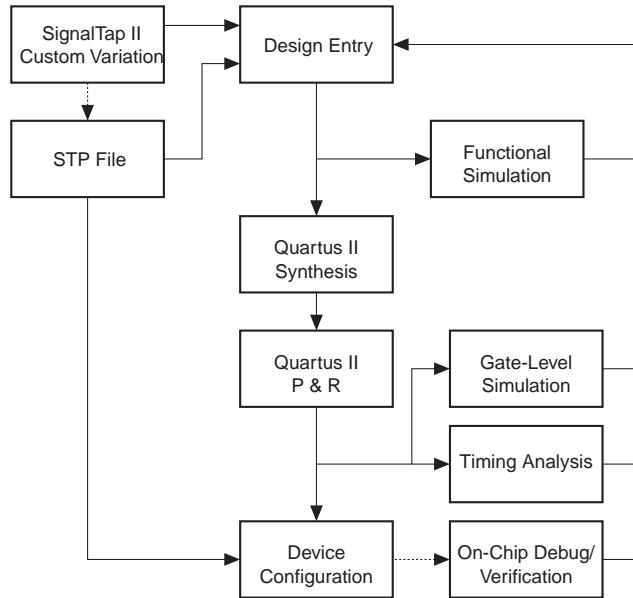
Feature	Benefit
Up to 1,024 channels in each device	Enables you to sample many signals and wide bus structures
Up to 128K samples in each device	Enables you to capture a large sample set for each channel
clock frequencies up to 200 MHz	Allows you to sample data at up to 200 MHz
Resource usage estimator	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 free with your Quartus II subscription

The SignalTap II Logic Analyzer supports the following device families: Stratix® II, Stratix, Stratix GX, Cyclone™ II, Cyclone, APEX™ II, APEX 20KE, APEX 20KC, APEX 20K, Excalibur™, and Mercury™.

Including the SignalTap II Logic Analyzer in Your Design

There are two ways to use the SignalTap II Logic Analyzer. The first method involves creating a SignalTap II file (.stp) and then configuring the details of the STP file. The second method involves creating and configuring the STP file with the MegaWizard® Plug-In Manager and then instantiating the hardware description language (HDL) output module from the MegaWizard Plug-In Manager in your HDL code.

Figure 10–2 illustrates the process of setting up and using the SignalTap II Logic Analyzer using both methods. The diagram shows the flow of operations from the initial MegaWizard Plug-in Manager custom variation to the final device configuration.

Figure 10–2. SignalTap II Flow

Using the STP File to Create an Embedded Logic Analyzer

In order to create an embedded logic analyzer, you can use an existing STP file or create a new STP file.

Creating an STP File

The STP file contains the SignalTap II Logic Analyzer settings and the captured data for viewing and analysis. To create a new STP file, follow these steps:

1. In the Quartus II software, choose **New** (File menu).
2. Click on the **Other Files** tab and select **SignalTap II File**.
3. Click **OK**.

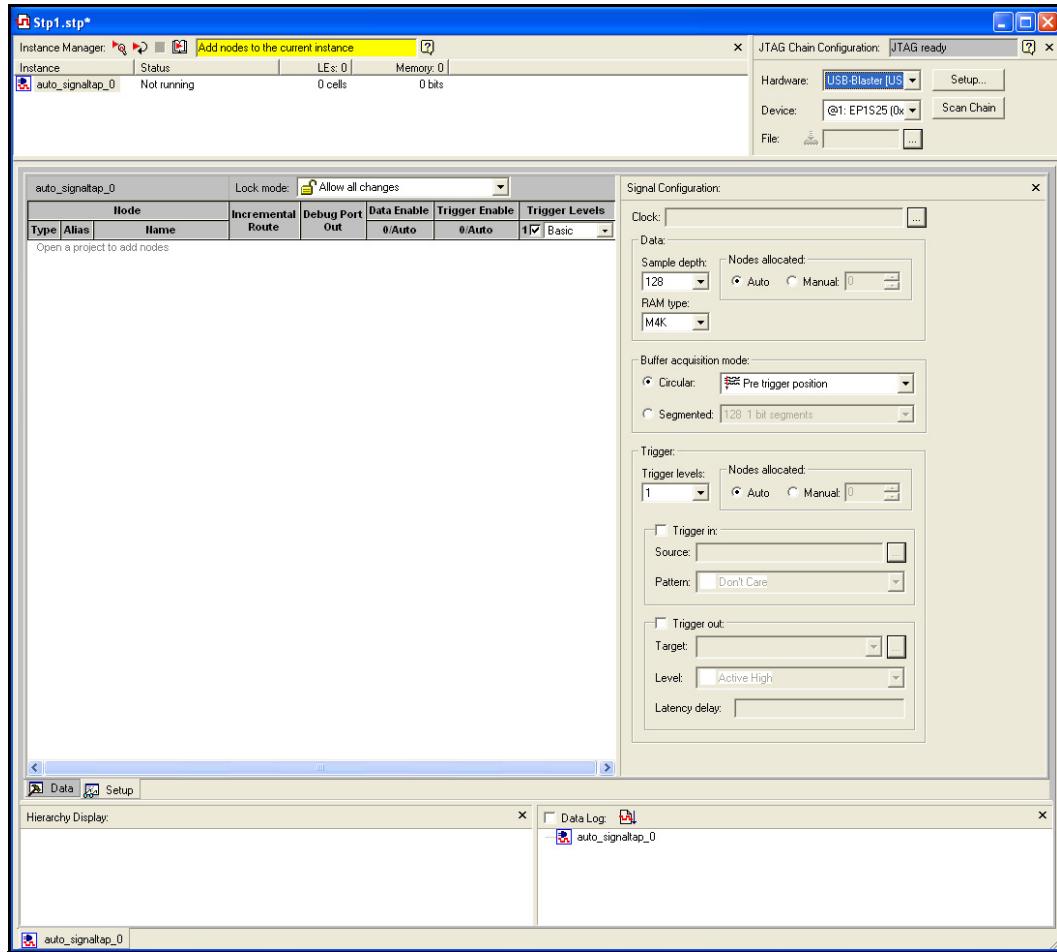
To open an existing STP file, choose **SignalTap II Logic Analyzer** (Tools menu). If no STP file exists for the current project this method is also used to create a new STP file.



Alternately, you can open a file by choosing **Open** (File menu).

Both of these methods open the SignalTap II window. See [Figure 10–3](#).

Figure 10–3. SignalTap II Window



Assigning an Acquisition Clock

You must assign a clock signal to control the acquisition of data by the SignalTap II Logic Analyzer. The acquisition clock samples data on every rising edge. You can use any signal in your design as the acquisition clock. However, for best results, Altera® recommends using a global clock for data acquisition and not a gated clock. 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 Timing Analyzer displays 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** list to open the Node Finder.
3. Select **SignalTap II: pre-synthesis** in the **Filter** list.



You cannot insert a clock signal that appears in the **SignalTap II: Post-fitting filter**.

4. In the **Named** box, enter the name of the signal that you would like to use as your sample clock.
5. To start the node search, click **List**.
6. In the **Nodes Found** list, select the node representing the design's global clock signal.
7. To copy the selected node name to the **Selected Nodes** list, click ">".
8. Click **OK**.
9. The node is now specified as the clock in the **SignalTap II** window.

If you do not assign an acquisition clock in the **SignalTap II** window, 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 on your PCB drives the acquisition clock.

Assigning Signals to the STP File

You can assign the following two types of signals to your STP file:

- Pre-synthesis—A pre-synthesis signal exists after design elaboration, but before any synthesis optimizations are done by physical synthesis. This set of signals should reflect your register transfer Level (RTL) signals.

- Post-fitting—A post-fitting signal exists after physical synthesis optimizations and place-and-route.



To add only pre-synthesis signals to your STP file, choose **Start Analysis & Elaboration** (Processing menu). This is particularly useful if you want to quickly add a new node after you have made design changes.

After a successful Analysis and Elaboration, the signals shown in red text are invalid signals and you must remove them from the STP file to ensure correct operation. The SignalTap II Health Monitor also indicates that an invalid node name exists in the STP file.

Assigning Data Signals

To assign data signals, follow these steps:

1. Perform Analysis and Elaboration, or Analysis and Synthesis, or compile your design.
2. In the SignalTap II Logic Analyzer window, click the **Setup** tab.
3. Double-click in the STP window to launch the Node Finder.
4. Select **SignalTap II: pre-synthesis** or **SignalTap II: post-fitting** in the **Filter** list.
5. In the **Named** box, enter a node name, partial node name, or 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 STP file.
7. To copy the selected node names to the **Selected Nodes** list, click “>”.
8. To insert the selected nodes in the STP file, click **OK**.



Alternately, you can drag and drop signals from the Nodefinder into an STP file.

Specifying the Sample Depth

The sample depth specifies the number of samples that are stored for each signal. To set the sample depth, select the desired number of samples in the **Sample Depth** list. The sample depth ranges from 0 to 128K samples.

Triggering the Analyzer

To control how the analyzer is triggered, set the trigger type and number of trigger levels:

Trigger Type: Basic or Advanced

If **Trigger Type** is set to **Basic**, you must set the **Trigger Pattern** for each signal in the STP file. You can set the **Trigger Pattern** to any of the following conditions:

- Don't Care
- Low
- High
- Falling Edge
- Rising Edge
- Either Edge

Data capture begins when the logical AND of all the signals for a given level evaluates to TRUE.

If **Trigger Type** is set to **Advanced**, you must build an expression that is used to trigger the analyzer.



For more information on trigger types, see “[Creating Complex Triggers](#)” on page 10–16.

Number of Trigger Levels

The multiple trigger level feature gives you precise control over the trigger condition that you build. This allows for more complex data capture commands to be given to the logic analyzer, providing greater accuracy and problem isolation. You can create up to ten trigger levels.

The SignalTap II Logic Analyzer first evaluates the trigger patterns associated with trigger level 1. When the expression for trigger level 1 evaluates to TRUE, SignalTap II Logic Analyzer evaluates the expression for trigger level 2. This process continues until all trigger levels are processed and the final trigger level evaluates to TRUE.

The multiple trigger level feature can be used with Basic Triggers or Advanced Triggers.

Select the desired number of trigger levels in the **Trigger Levels** list.

You can disable the ability of a signal to trigger the analyzer by disabling the **Trigger Enable** for that signal in the STP file. This option is useful when you only want to see captured data for a signal, and are not using that signal as a trigger.

You can disable 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.

Specifying the Trigger Position

You can specify the amount of data that is acquired before the trigger event. Select the desired ratio of pre-trigger data to post-trigger data by selecting 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)
- Continuous — This selection saves signal activity indefinitely (until stopped manually)

After you configure the STP file, you must compile it with your Quartus II project before you can use it to analyze your design.

Compiling Your Design with SignalTap II Logic Analyzer

The first time you create and save an STP file, the Quartus II software automatically adds the file to your project. However, you can add an STP file manually by performing the following steps:

1. Choose **Settings** (Assignments menu).
2. In the **Category** list, select **SignalTap II Logic Analyzer**.
3. Turn on **Enable SignalTap II Logic Analyzer**.
4. In the **SignalTap II File Name** box, type the name of the STP file you want to compile, or select a file name with **Browse**.
5. Click **OK**.
6. To begin the compilation, select **Start Compilation** (Processing menu).



When you compile your design with an STP file, the **sld_singaltap** and **sld_hub** entities are added in the compilation hierarchy. These two entities are the main components of the SignalTap II Logic Analyzer.

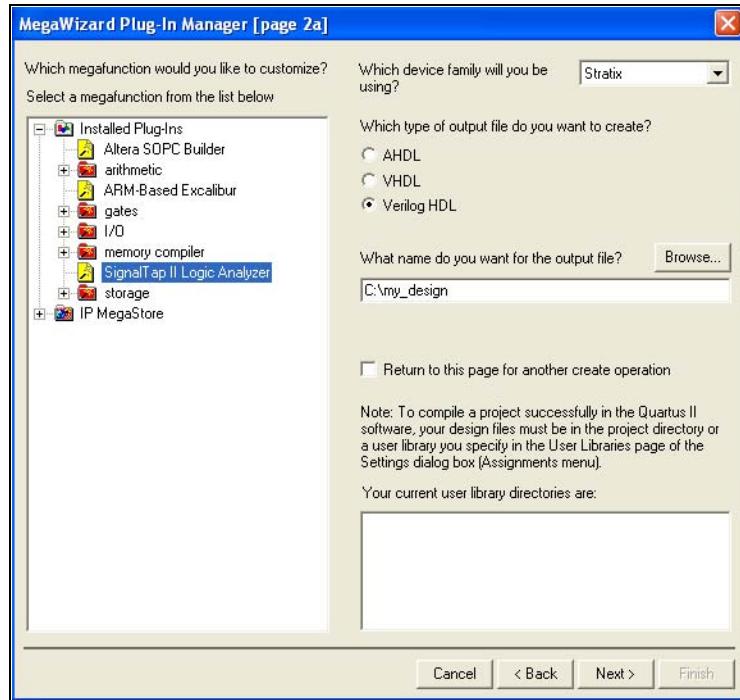
Using the MegaWizard Plug-In Manager to Create Your Embedded Logic Analyzer

Alternatively, you can create a SignalTap II Logic Analyzer 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, along with using the method described in “[Using the STP File to Create an Embedded Logic Analyzer](#)” on page 10–4.

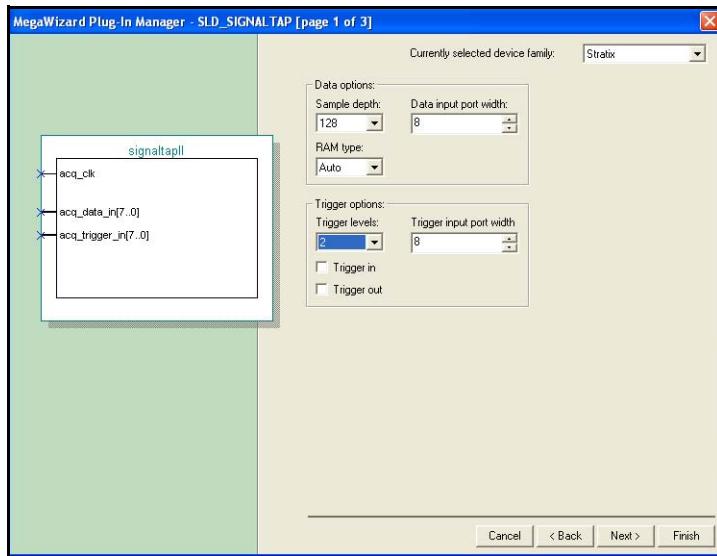
Creating the HDL Representation of the SignalTap II Logic Analyzer

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, follow these steps:

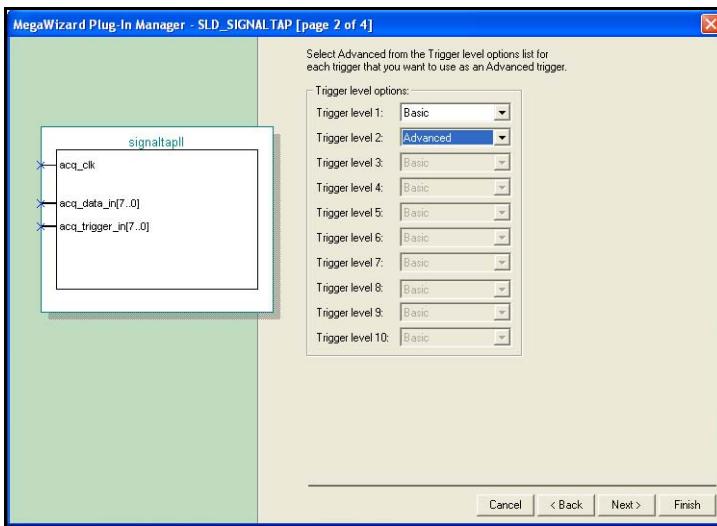
1. Launch the MegaWizard Plug-In Manager by choosing **MegaWizard Plug-In Manager** (Tools menu) in the Quartus II software.
2. Select **Create a new custom megafunction variation**.
3. Click **Next**.
4. Choose the **SignalTap II Logic Analyzer**. Select an output file type and enter the desired name of the SignalTap II megafunction. You can choose AHDL (.tdf), VHDL (.vhd), or Verilog HDL (.v) as the output file type.
5. Click **Next**. See [Figure 10–4](#).

Figure 10–4. Select an Output File & Enter the Selected SignalTap II Name

6. Configure the analyzer by specifying the **Sample Depth**, **Memory Type**, **Data Input Width**, **Trigger Input Width**, and **Number of Trigger Levels**.
7. Click **Next**. See [Figure 10–5](#).

Figure 10–5. Select the Parameters for the Analyzer

- Set the Trigger level options by choosing **Basic** or **Advanced**. See Figure 10–6.

Figure 10–6. Basic & Advanced Trigger Options

- Click **Finish** to complete the process of creating an HDL representation of the SignalTap II Logic Analyzer.

SignalTap II Megafunction Ports

Table 10–2 provides information on the SignalTap II megafunction ports.



Refer to the latest version of the Quartus II Help for the most current information on the ports and parameters for this megafunction.

Table 10–2. SignalTap II Megafunction Ports			
Port Name	Type	Required	Description
acq_data_in	Input	No	This set of signals represent the 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

The process of instantiating the logic analyzer in your HDL is similar to instantiating any other Verilog HDL or VHDL megafunction in your design. You can instantiate up to 127 analyzers in your design or as many as will physically fit in the FPGA. Once you have instantiated the SignalTap II file in your HDL, compile your Quartus II project to fit the logic analyzer in the target FPGA.

To capture and view the data, you must create an STP file from your SignalTap II HDL output file. The STP file is automatically created for you when you select **Create SignalTap II File from Design Instance(s)** from the **Create/Update Menu** (File menu).

Programming the Device for SignalTap II Analysis

When the compilation is complete, you must program the FPGA. To program a device for use with the SignalTap II Logic Analyzer, follow these steps:

- In the **JTAG Chain Configuration panel in the STP file**, select the SRAM Object File (.sof) that includes the SignalTap II Logic Analyzer.

2. Click **Scan Chain**.
3. In the **Device** list, select the device to which you want to download the design.
4. Click **Program Device**.

View Data Samples

To capture and view data samples, follow these steps:

1. Select the **Run** button.
2. Run the SignalTap II Logic Analyzer by clicking **Run** or **AutoRun** in the **SignalTap II** window.

Data capture begins when the trigger event evaluates to TRUE.



For more information on triggering, see the “Triggering the Analyzer” section.

The SignalTap II toolbar has four options for running the analyzer:

- **Run**—SignalTap II Logic Analyzer runs until the trigger event occurs. When the trigger event occurs, data capture stops.
- **Stop**—SignalTap II analysis stops. The acquired data does not appear if the trigger event has not occurred.
- **AutoRun**—SignalTap II Logic Analyzer continuously captures data until the **Stop** button is clicked.
- **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.

Advanced Features

This section describes the following advanced features:

Section	Page
Preserving FPGA Memory	10-15
Creating Complex Triggers	10-16
Using External Triggers	10-19
Embedding Multiple Analyzers in One FPGA	10-22
Faster Compilations.	10-23
Time Bars & Next Transition	10-24
Saving Captured Data	10-25
Converting Captured Data to Other File Formats	10-25
Creating Mnemonics for Bit Patterns	10-25
Capturing Data to a Specific RAM Type	10-27
FPGA Resources Used by SignalTap II.	10-27
Using SignalTap II in a Lab Environment	10-28
Remote Debugging Using the SignalTap II Logic Analyzer	10-29
Signal Preservation	10-32
Tappable Signals	10-33
Timing Preservation with SignalTap II Logic Analyzer . .	10-33
Using SignalTap II Logic Analyzer to Simultaneously Debug Multiple Designs	10-34
Locating a Node in the Chip Editor	10-34

Preserving FPGA Memory

You can configure the SignalTap II Logic Analyzer to store captured data in the device RAM, or route captured data to I/O pins to analyze with an external logic analyzer. The following factors can affect the mode of operation you choose:

- The availability of device RAM and I/O pins
- The number of trigger levels being used in analysis
- Whether the SignalTap II Logic Analyzer is used in conjunction with external test equipment

When device RAM is limited, the software can route internal signals to unused I/O pins for capture by an external logic analyzer. This method is useful for memory-intensive applications in which the amount of saved data exceeds the available sample buffer depth provided by the device RAM. The Quartus II software automatically generates debugging port signals that connect internal FPGA signals to output pins. You must assign these signals to I/O pins. To use the SignalTap II Logic Analyzer debugging port configuration, follow these steps:

1. Click on a signal in the **Debug Port Out** column.

2. Choose **Enable Debug Port** (**Edit menu**).

If you want to rename the debugging port pin, type the new name in the **Out** column. The default signal name for the debugging ports is `auto_stp_debug_out_<m>_<n>`, where *m* refers to the instance number and *n* refers to the signal number.

3. Manually assign the debugging port signal name to an unused I/O pin.

Creating Complex Triggers

The most crucial feature of an analyzer is the triggering capability. If you do not have the ability to create a trigger condition that allows you to capture relevant data, your logic analyzer may not help you debug your design.

With the SignalTap II Logic Analyzer, you can build very complex triggers that allow you to capture data when a set of trigger conditions exist. Advanced triggers are built with a simple graphical interface. You can drag-and-drop operators into the Advanced Trigger window to build the complex trigger condition in an expression tree. The operators that you can use are listed in [Table 10–3](#).

Table 10–3. Advanced Triggering Operators [Note \(1\)](#) (Part 1 of 2)

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

Table 10–3. Advanced Triggering Operators *Note (1)* (Part 2 of 2)

Name of Operator	Type
Bitwise Complement	Bitwise
Bitwise AND	Bitwise
Bitwise OR	Bitwise
Bitwise XOR	Bitwise
Edge and Level Detector	Signal Detection
Continuous	Counter
State	Counter
Event	Counter

Note to Table 10–3:

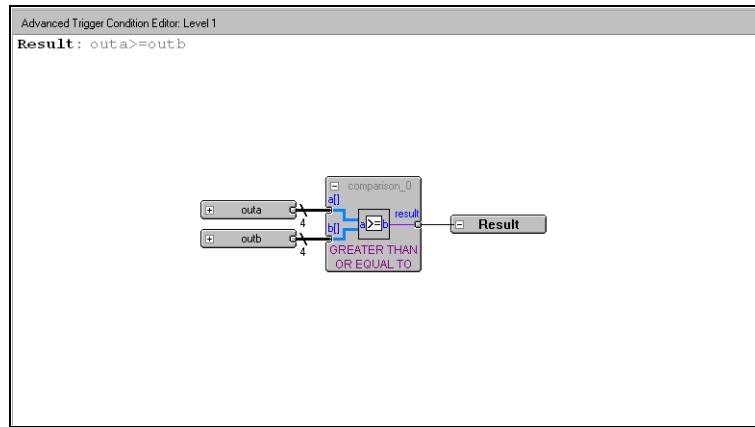
- (1) For more information on each of these operators, see Quartus II Help.

You can configure some of the operators at run time. This allows you to change one operator type to another operator type without recompiling your design. Operators that have a white background are run-time configurable.

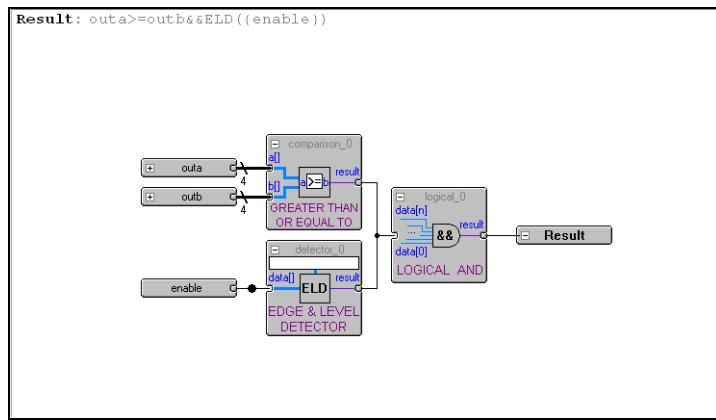
Adding many objects to the Advanced Trigger Condition Editor can make the workspace cluttered and unreadable. Altera recommends that you use the **Arrange All Objects** from the right button pop-up menu when building your advanced trigger condition. You can also use the **Zoom-Out** command to fit more objects into the Advanced Trigger Condition editor.

The following examples show how to use Advanced Triggering:

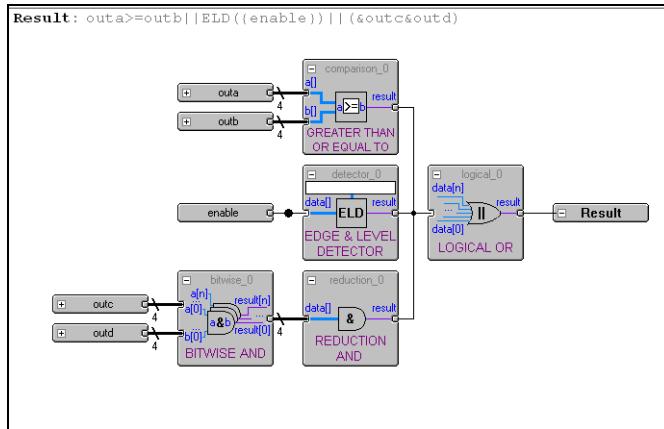
- Trigger when bus `outa` is greater than or equal to `outb` (see Figure 10–7).

Figure 10–7. Bus outa is Greater Than or Equal to outb

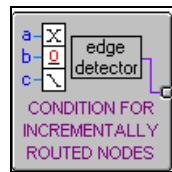
- Trigger when bus `outa` is greater than or equal to `outb`, and when the enable signal has a rising edge (see [Figure 10–8](#)).

Figure 10–8. 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 with bus `outc` and bus `outa`, and all bits of the result of that operation are 0 (see [Figure 10–9](#)).

Figure 10–9. Bitwise AND Operation

You can use the advanced triggering capability only with pre-synthesis nodes. Post-fitting nodes are used for basic trigger operations only. However, you can create an advanced trigger that uses the results of the basic trigger created with post-fitting nodes as an element of an advanced trigger condition. When your STP file contains post-fitting nodes, the symbol (shown in [Figure 10–10](#)) appears in the advanced trigger panel.

Figure 10–10. Symbol for STP File Containing Post-Fitting Nodes

The output of this node can be combined with the operators listed in [Table 10–3](#).

Using External Triggers

You can create a trigger input that allows you to trigger the SignalTap II Logic Analyzer from an external source. The analyzer can also operate in the trigger output configuration in which it supplies an external signal to trigger other devices. These features allow you to synchronize the internal logic analyzer with external logic analysis equipment.

Trigger In

To use Trigger In, perform the following steps:

1. In the SignalTap II window, click the **Setup** tab.
2. In the **Signal Configuration** window, turn on **Trigger In**.
3. In the **Pattern** pull down list, select the condition you would like to act as your trigger event.
4. Click on the **Browse** button next to **Trigger In**.

When the **Node Finder** window appears, select the signal (either an input pin or an internal signal) that you want to drive the Trigger In source.

Trigger Out

To use Trigger Out, perform the following steps:

1. In the SignalTap II window, click the **Setup** tab.
2. In the **Signal Configuration** window, turn on **Trigger Out**.
3. In the **Level** list, select the condition you would like to signify that the trigger event is occurring.
4. Click **Browse** next to the **Trigger Out**.

When the **Node Finder** window appears, select the signal (either an output pin or an internal signal) that you want to drive the Trigger Out.

Using the Trigger Out of One Analyzer as the Trigger In of Another Analyzer

One 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 first analyzer and set the name for the Trigger Out signal (see the colored portion of [Figure 10–11](#)). Next, you must enable the **Trigger In** of the

second analyzer and set the name of the Trigger In of the second analyzer as the Trigger Out of the first analyzer (see the colored portion of Figure 10–12).

Figure 10–11. Enabling the Trigger Out Signal

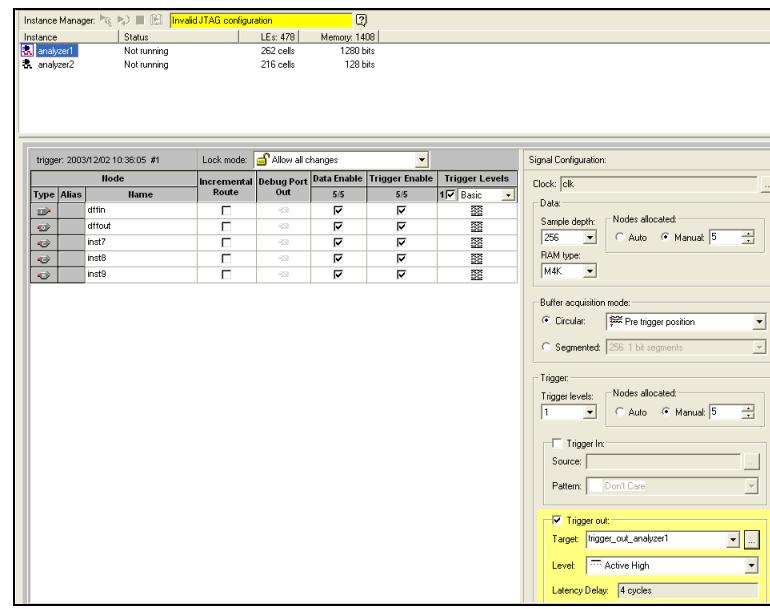
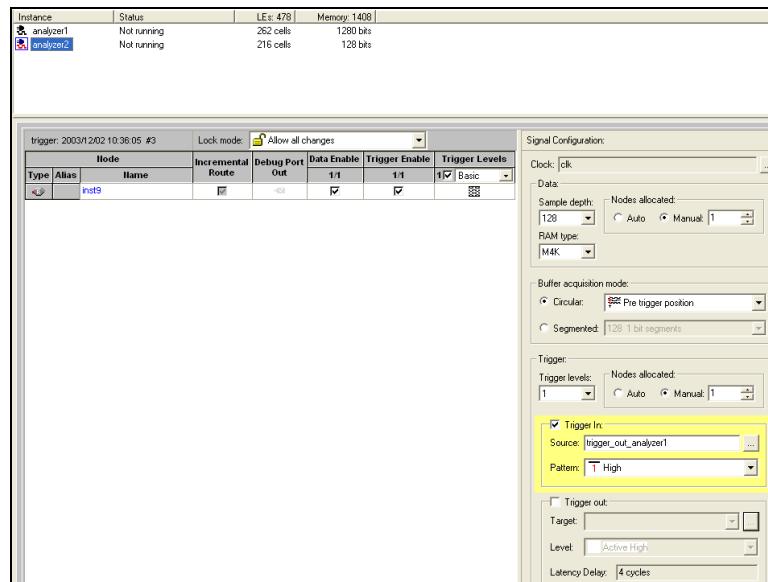


Figure 10–12. Enabling the Trigger In Signal

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 STP file, the LE count 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, while another set of signals in the same clock domain need a sample depth of 1K, you can create two instances to meet these needs.

To create multiple analyzers, select **Create Instance** (**Edit menu**), or right-click in the **Instance Manager** window, and select **Create Instance**.



You can start all instances at the same time by selecting all instances and clicking **Run** in the SignalTap II toolbar.

Faster Compilations

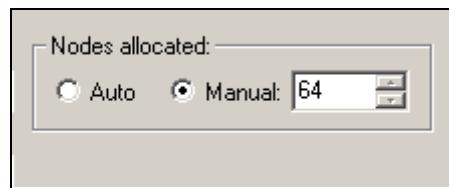
Making significant changes such as expanding your sample depth or adding new signals to the SignalTap II Logic Analyzer could potentially force a recompilation of your design. To prevent a full recompilation, the SignalTap II Logic Analyzer includes a feature called Incremental Routing. This feature enables you to add new signals or swap existing signals for entirely new signals without performing a full recompilation. Adding new signals to your STP file does not affect the placement of your existing design in the Quartus II software.

Adding New Nodes Without Performing a Full Recompilation

If you plan to use this feature, you must estimate the number of signals you eventually plan to add to your STP file. Once you have estimated the number of nodes, click **Manual** and specify the number of signals in the **Nodes Allocated** dialog box. For example, if the current configuration of your STP file has 32 signals, but you plan to expand this to 64 signals, you must allocate 64 signals manually when you first create your STP file in order to avoid a full recompilation. In this example the Quartus II fitter allocates the extra 32 signals as place-holders.

Setting **Nodes Allocated** to **Auto** causes the Quartus II software to build the SignalTap II Logic Analyzer to accommodate only the number of channels that were selected in the STP file.

Figure 10–13. Nodes Allocated



Replacing Existing Signals With New Signals Without Performing a Full Recompilation

As shown in [Figure 10–14](#), the **SignalTap II Setup** window shows pre-synthesis nodes and post-fitting nodes, and an **Incremental Route** column. Post-fitting nodes are displayed in blue, with the **Incremental Route** option enabled and dimmed, so it cannot be edited.

By turning on **Incremental Routing** for pre-synthesis nodes, you preserve the signal to the post-fitting stage of the compilation. You can later delete the incrementally-routed pre-synthesis node and replace it with a post-fitting node. You cannot replace this node with a SignalTap II pre-synthesis node.

Figure 10–14. The SignalTap II Setup Window Note (1)

Type	Alias	Name	Incremental Route	Debug Port Out	Data Enable	Trigger Enable	Trigger Levels
					68/Auto	68/Auto	1 <input checked="" type="checkbox"/> Advanced <input type="button" value="▼"/>
		inf1_b	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	XXXXXXXXXXXX...
		XX	<input type="checkbox"/>		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
		outff_a	<input type="checkbox"/>		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
		out	<input type="checkbox"/>		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
		a[1]	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
		a[0]	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	

Note to Figure 10–14:

- (1) Post-fitting nodes are displayed in blue, and Incremental Route is always turned on.

The next time you add a SignalTap II post-fitting node to the STP file and start a compilation, the Quartus II software incrementally routes only the new nodes. When the Quartus II software performs incremental routing, the existing placement and routing of your design is not modified.

If routing resources are limited, the Quartus II software may not be able to incrementally route your SignalTap II signal. If you encounter a situation in which, the Quartus II software is not able to route your signal turn on the **Modify latest fitting result during a SignalProbe Compilation** option. When this option is turned on, the placement and routing of your existing design may change.

Time Bars & Next Transition

Time bars enable you to calculate the number of clock cycles between two transitions for 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. The captured data has only one master time bar; however, you can create an unlimited number of reference time bars that display the time relative to the master time bar.
- **Reference Time Bar**—The Reference Time Bar’s label displays 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 feature is very useful when the sample depth is very large and the rate at which signals toggle is very low.

Saving Captured Data

The data log shows the history of captured data that is acquired with the SignalTap II Logic Analyzer and the triggers used to capture the data. The analyzer acquires data, stores it in a log, and displays it as waveforms. The default name for the log is based on the timestamp that shows when the data was acquired. It is a good idea to 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. To enable data logging, turn on the **Data Log** option. To recall a data log for a given trigger set and make it active, double click on the name of the data log in the list.



This feature is useful for organizing different sets of trigger conditions and different sets of signal configurations.

Converting Captured Data to Other File Formats

You can export captured data in the following industry-standard file formats, some of which can be used with other EDA simulation tools:

- Comma Separated Values (.csv) File
- Table (.tbl) File
- Value Change Dump (.vcd) File
- Vector Waveform File (.vwf)
- Graphics Format (.jpg, .bmp) File

To export the SignalTap II Logic Analyzer's captured data, choose **Export** (File menu) and specify the **File Name**, the **Export Format**, and the **Clock Period**.

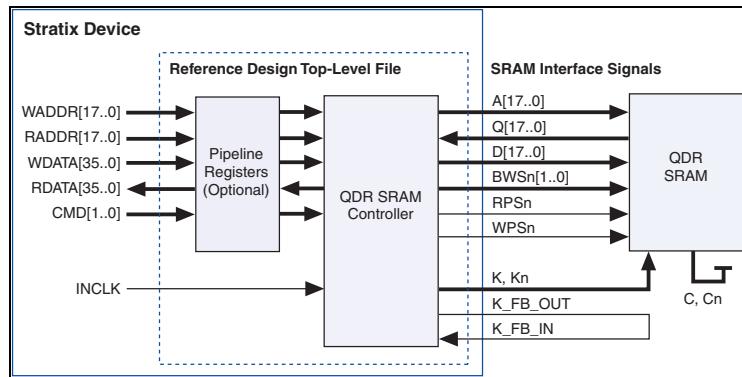
Creating Mnemonics for Bit Patterns

The mnemonic table feature allows you to assign a meaningful name to a set of bit patterns. To create a mnemonic table, right-click in the **Setup** view of an STP file and select **Mnemonic Setup**. To assign a group of signals to a mnemonic value, right-click on the group, and select **Bus Display Setup** (right button pop-up menu).

Segmenting Your Sample Depth

The Buffer Acquisition feature in SignalTap II Logic Analyzer allows you to significantly reduce the amount of memory that is required for SignalTap II data acquisition. This feature makes it easier to debug systems that contain relatively infrequent periodic events. An example of this type of system is shown in Figure 10–15.

Figure 10–15. Example System Generating Periodic Events



The SignalTap II Logic Analyzer can be used to verify functionality of the design shown in Figure 10–15 to ensure that the correct data are written to the SDRAM controller. The buffer acquisition in the SignalTap II Logic Analyzer allows you to monitor the RDATA port when H' 0F0F0F0F is sent into the RDADDR port. You can monitor multiple read transactions from the SDRAM device without rerunning the SignalTap II Logic Analyzer. 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 through the Signal Configuration settings.

To enable and configure buffer acquisition, select **Segmented** in the **SignalTap II** window, and then choose the number of segments to use. Selecting sixty-four 64-bit segments allows you to capture 64 read cycles when the RADDR signal is H' 0F0F0F0F.



For more information on the buffer acquisition mode, see *Setting the Buffer Acquisition Mode* in the Quartus II Help.

Capturing Data to a Specific RAM Type

When using the SignalTap II Logic Analyzer with a Stratix device, you can select the RAM type that is used to store the acquisition data. 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 may be ideal to place this application into M-RAM blocks so that the remaining M512 or M4K blocks can be used for SignalTap II data acquisition.

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 Stratix FPGA. For example, there are 94 M512 RAM blocks in a Stratix EP1S10 device. For 94x576 RAM bits, if you set the RAM type to M512, then you should make sure that your SignalTap II configurations do not need more than the number of RAM bits that are available for that type of memory.

FPGA Resources Used by SignalTap II

The SignalTap II Logic Analyzer has a built-in resource estimator that dynamically calculates the number of LEs and the amount of memory that each SignalTap II analyzer uses. This feature is useful when device resources are limited and you must know what device resources the SignalTap II analyzer uses. The value reported in the resource usage estimator may vary by as much as 5% from the actual resource usage.

The following table provides an estimate of the number of LEs and the amount of memory that is required to add SignalTap II Logic Analyzer to your design.

Table 10–4. *SignalTap II Logic Analyzer M4K Block Utilization for Cyclone, Stratix GX, & Stratix Devices*

Note (1)

Signals (Width)	Samples (Width)			
	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 10–4

- (1) When configuring a SignalTapII Logic Analyzer, the Instance Manager reports an estimate of the memory bits and logic elements required to implement the given configuration.

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 lab environments where you may not have a workstation that meets the requirements for a complete Quartus II installation or 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 web site. The stand alone version can be downloaded at https://www.altera.com/support/software/download/programming/quartus2/dnl-quartus2_programmer.jsp.

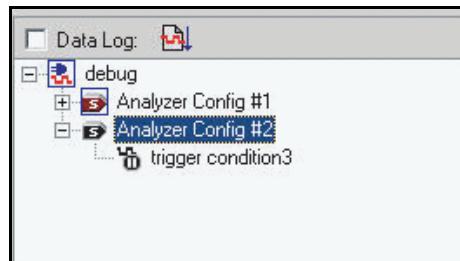
Managing Multiple STP Files

Many times you have more than one STP file to debug in one design. Each STP file potentially has a different group of signals; these groups of signals allow you to debug different blocks in your design. Along with each STP file there is an associated programming file (SOF). Managing all of the STP files and their associated programming file is a challenging task. To help you manage these files, you can use the **Data Log** feature and the **SOF Manager**.

The **Data Log** allows you to store multiple STP configurations. In **Figure 10–16**, there are two configurations in one STP file. You can toggle between the active configurations by double clicking 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 STP file. The active configuration that is displayed in the STP file is indicated by the blue square around the signal set in the Data log.

Figure 10–16. Data Log



The SOF Manager allows you to embed multiple SOF files into one STP file. To embed a new SOF file in the STP file choose **Attach SOF file** (right button pop-up menu) in the SOF Manager. See [Figure 10–17](#).

Figure 10–17. SOF Manager



As you toggle between configurations in the Data Log, you can extract the SOF file associated with that particular configuration and use the programmer in the SignalTap II Logic Analyzer to download the new SOF to the FPGA.

Remote Debugging Using the SignalTap II Logic Analyzer

You can use a SignalTap II Logic Analyzer to debug a design that is running on a PCB in a remote location.

To perform this debugging session you need the following setup:

- Quartus II software installed on the local PC
- Stand-alone SignalTap II Logic Analyzer installed on the remote PC
- Programming hardware connected to the PCB at the remote location
- TCP/IP connection

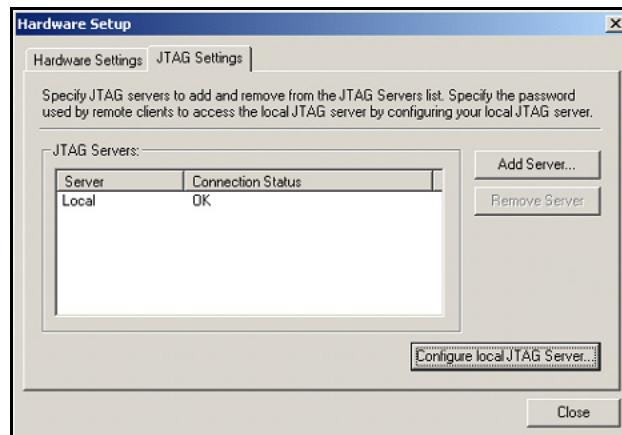
Equipment Setup:

1. On the PC in the remote location install the stand-alone version of the SignalTap II Logic Analyzer. This remote computer must have Altera programming hardware connected, for example, USB-BlasterTM or ByteBlasterTM.
2. 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 in the Remote PC

1. Select **Hardware Setup** from the Quartus II programmer.
2. Select the **JTAG Settings** tab. See [Figure 10–18](#).
3. Click the **Configure local JTAG server** button.

Figure 10–18. Configure Hardware Settings



4. In the **Configure Local JTAG Server** dialog box (see [Figure 10-19](#)), turn on **Enable remote clients to connect to the local JTAG server** and type in your password. Type your password again in the **Confirm Password** box and click **OK**.

Figure 10-19. Configure Local JTAG Server



Software Setup in the Local PC

1. Launch the Quartus II programmer.
2. Click **Hardware Setup**.
3. Click the **JTAG settings** tab. Click **Add server**.
4. In the **Add Server** dialog box (see [Figure 10-20](#)), type the network name or IP address of the server you want to use and the password for the JTAG server created on the Remote PC.

Figure 10-20. Add Server Dialog Box



5. Click **OK**.

SignalTap II Setup in the Local PC

1. Select the hardware by clicking the **Hardware Setup** tab and choose the hardware on the Remote PC. See [Figure 10–21](#).

Figure 10–21. SignalTap II Hardware Setup



2. Click **Close**.
3. Program the PCB in the remote location using the TCP/IP link and the hardware on the remote PC.

Signal Preservation

Many of your RTL signals are optimized during the process of synthesis and place-and-route. This may lead to issues when you debug your design, because the post-fitting signal names differ significantly from your RTL names. 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. Therefore, you could see an increase in resource utilization or a decrease in timing performance. The two attributes you could use are:

- **Keep** — This attribute ensures that combinational signals are not removed
- **Preserve** — This attribute ensures that registers are not removed



For more information on using these attributes, see the *Quartus II Integrated Synthesis* chapter in Volume 1 of the *Quartus II Handbook*.

Tappable Signals

Not all of the post-fitting signals in your design are available in the **SignalTap II: post-fitting** in the Node Finder. Listed below are the signal types that cannot be tapped:

- 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 LE.
- PLL `clkout`—You cannot tap the output clock of a PLL. Due to architectural restrictions, the clock out signal can only feed the clock port of a register.
- JTAG Signals—You cannot tap the JTAG (`TCK`, `TDI`, `TDO`, and `TMS`) signals.
- LVDS—You cannot tap the data out from a SERDES block.

Timing Preservation with 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, you are adding IP to your existing design, therefore you could potentially 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 back-annotate your design prior to inserting the SignalTap II Logic Analyzer. This allows you to run your design at the desired frequency.

In addition to back-annotating your design, you can use the following techniques to help maintain timing:

- Avoid adding critical path signals to your STP file.
- Minimize the number of signals that have Trigger Enable selected. By default, all of the signals that you add to the STP file have the Trigger Enable turned on. Turn off Trigger Enable for signals you do not plan to use as triggers.
- Use the Basic Trigger whenever possible. Using the Advanced Trigger increases the amount of logic, which may add extra delay to your circuit.
- Minimize the number of combinational signals you have added to your STP file. Add registers whenever possible.
- Specify an f_{MAX} constraint for each clock in your design.
- Back-annotate your design prior to compiling with the SignalTap II Logic Analyzer.



For an example of timing preservation with the SignalTap II Logic Analyzer, see the *Design Optimization for Altera Devices* chapter in Volume 2 of the *Quartus II Handbook*.

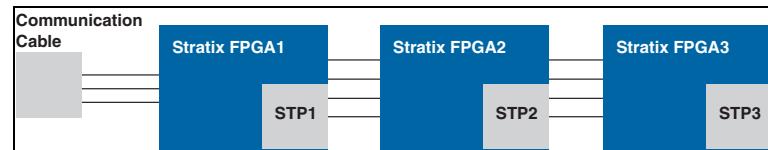
Using SignalTap II Logic Analyzer to Simultaneously Debug Multiple Designs

You can simultaneously debug multiple designs using one instance of the Quartus II software. To perform this operation, follow these steps:

1. Create, configure, and compile the STP file for each design.
2. Open each individual STP file. Note: you do not have to open a Quartus II project to open an STP file.
3. Use the JTAG Chain controls to select the target device in each STP file.
4. Program each FPGA.
5. Run each analyzer independently.

Figure 10–22 shows a JTAG chain and its associated STP files.

Figure 10–22. JTAG Chain



Locating a Node in the Chip Editor

Once you have found the source of a bug in your design using the SignalTap II Logic Analyzer, you can locate that signal in the Chip Editor. Once in the Chip Editor, you can modify your design to correct the flaw that was found using the SignalTap II Logic Analyzer. To locate a signal from the SignalTap II Logic Analyzer in the Chip Editor, right-click on a signal in the STP file and select **Locate in Chip Editor**.



For more information on using the Chip Editor, see the *Design Analysis & Engineering Change Management with Chip Editor* chapter in Volume 3 of the *Quartus II Handbook*.

SignalTap II Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. 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 `quartus_sh --qhelp` at a system command prompt to run the Quartus II Command-Line and Tcl API Help utility.



For more information on Quartus II scripting support, including examples, refer to the *Tcl Scripting and Command-Line Scripting* chapters 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. The following options helps you better understand how to use the `quartus_stp` command:

Table 10-5. SignalTap II Command Line Options

Option	Usage	Description
<code>stp_file</code>	<code>quartus_stp --stp_file [<stp_filename>]</code>	Assigns the specified STP file to the <code>USE_SIGNALTAP_FILE</code> in the Quartus II Settings File (QSF).
<code>enable</code>	<code>quartus_stp --enable</code>	Compiles the STP file specified in the QSF with your design. If no STP file is specified in the QSF, the <code>--stp_file</code> option must be used in conjunction with this.
<code>disable</code>	<code>quartus_stp --disable</code>	Removes the STP reference from the QSF. The SignalTap II Logic Analyzer is removed from the design database the next time you compile your design.
<code>verify_connections</code>	<code>quartus_stp --verify_connections</code>	Once a compilation with SignalTap II is complete, you need to use this option to ensure that all signals that you wanted to probe in your design were properly connected to the logic analyzer.

The following example illustrates how to compile a design with the SignalTap II Logic Analyzer using the command line:

```
quartus_stp --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
quartus_stp --verify_connections
```

The quartus_stp --stp_file stp1.stp --enable command sets the QSF variable and instructs the Quartus II software to compile the **stp1.stp** file with your design.

The quartus_stp --verify_connections command must be run after the quartus_fit command. The --verify_connections option verifies that all signals in the STP file were routed correctly to the logic analyzer.



For information on 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. To run Tcl file that has SignalTap II Tcl commands use the following command:

```
quartus_stp -t <Tcl file>
```

The available Tcl commands are listed in the table below:

Table 10–6. SignalTap II Tcl Commands (Part 1 of 2)

Command	Argument	Description
open_session	--name [stp_filename]	Opens the specified STP file. This file is where all captured data is stored.
run	--instance <instance_name> --signal_set <signal_set> --trigger <trigger_name> --data_log <data_log_name> (optional) --timeout <seconds> (optional)	Starts the analyzer. This command must be followed by all of the required arguments in order to properly start the analyzer. You can optionally specify the name of the data log you want to create. Also, you can optionally specify a timeout value to stop the analyzer if the trigger condition has not been met.

Table 10–6. SignalTap II Tcl Commands (Part 2 of 2)

Command	Argument	Description
run_multiple_start	None	Defines the start of a set of run commands. This command is used when multiple instances of data acquisition are started simultaneously. Add this command before the set of run commands that specify data acquisition. This command needs to be used with the run_multiple_end command. If the run_multiple_end is not included, the run commands do not execute.
run_multiple_end	None	Defines the end of a set of run commands. This command is used 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 STP file. The analyzer cannot be run once the STP file has been closed.
create_signaltap_hdl_file	None	Creates an STP file from SignalTap II megafunction created with the MegaWizard Plug-in Manager. You need to use the -stp_file command in order to properly create an STP file.



For more information on SignalTap II Tcl commands, please refer to the Quartus II on-line help.

The following example 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 into the data log.

```
#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 need to open the STP file that you used to capture data to examine the contents of the Data log.

Design Example: Preserving Timing

The following example shows the importance of back annotating your design prior to inserting the SignalTap II Logic Analyzer. The design files that are used for this example vary slightly from the FIR filter design that is included in the \qdesigns directory. To follow this example, you should first restore the `compile_fir_filter_original.qar` design file.

Scenario: After programming your FPGA you notice incorrect behavior with your circuit. Because you are using a fine-pitch package, using a traditional logic analyzer is not possible. To debug this design you need to use the SignalTap II embedded logic analyzer. The design calls for an f_{MAX} requirement of 125 MHz to be met.

1. Initial compilation without SignalTap II Logic Analyzer.

When you run the Quartus II Timing Analyzer, you see the following results for the main clock in the design (see [Table 10–7](#)).

Table 10–7. f_{MAX} Results from the Quartus II Timing Analysis

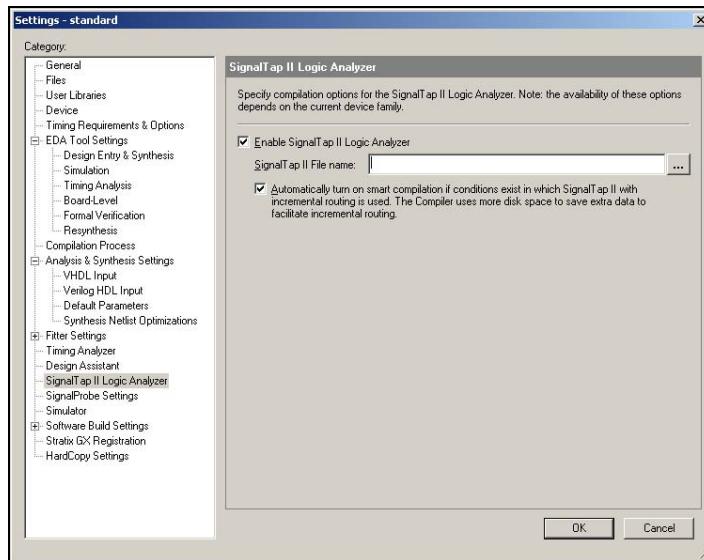
Slack (ns)	Actual f_{MAX} (MHz)	From	To	Clock Source
0.167	127.67	state_m:inst1 filter~22	acc:inst3 result[11]	clk
0.256	129.13	state_m:inst1 filter~22	acc:inst3 result[6]	clk
0.144	127.29	state_m:inst1 filter~22	acc:inst3 result[7]	clk
0.144	127.29	state_m:inst1 filter~22	acc:inst3 result[8]	clk

2. Compilation with the SignalTap II Logic Analyzer.

You are debugging this design with the SignalTap II Logic Analyzer, so you must compile it with an STP file. To enable the SignalTap II Logic Analyzer, the STP file included in the project archive (`stp1.stp`)

must be correctly set in the Quartus II software. Do this by enabling the STP file in the **SignalTap II Logic Analyzer** page of the **Settings** dialog box (Assignments menu), as shown in Figure 10–23.

Figure 10–23. Enabling the STP File in the SignalTap II Logic Analyzer Page



Once the compilation is complete, the Quartus II Timing Analyzer reports the results shown in Table 10–8.

Table 10–8. f_{MAX} Results from the Quartus II Timing Analysis with SignalTap II Logic Analyzer Added

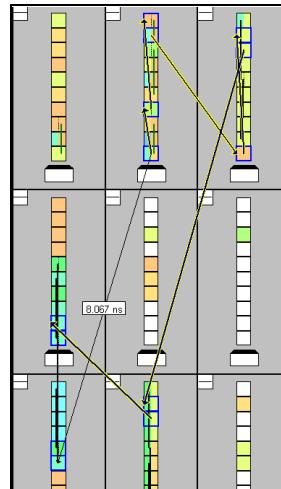
Slack (ns)	Actual f_{MAX} (MHz)	From	To	Clock Source
-0.266	120.98	state_m:inst1 filter~22	acc:inst3 result[11]	clk
-0.177	122.29	state_m:inst1 filter~22	acc:inst3 result[6]	clk
-0.076	123.82	state_m:inst1 filter~22	acc:inst3 result[7]	clk
-0.076	123.82	state_m:inst1 filter~22	acc:inst3 result[8]	clk



Notice that when you added the SignalTap II Logic Analyzer to your design, the longest register-to-register path changed. The delay increased by approximately 8%. This increase results in the system not meeting the timing requirements.

Figure 10–24 shows a failing path in the timing closure floorplan editor.

Figure 10–24. Failing Path in the Timing Closure Floorplan Editor



3. Back-annotate the original design.

To minimize the effect that the SignalTap II Logic Analyzer has on the original design, you should back-annotate the design to constrain it to a portion of the FPGA. You can do this by selecting **Back-Annotate Assignments** (Assignments menu). After you have back-annotated your design, it is safe to insert the SignalTap II Logic Analyzer into your project.

Compile the design and you see the results shown in Table 10–9.

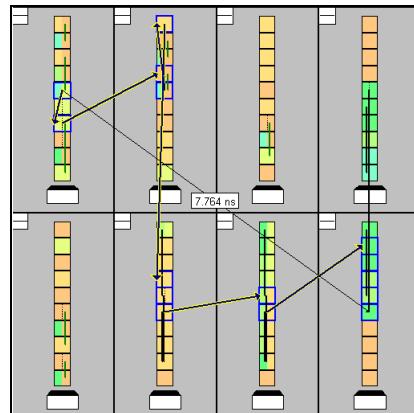
Table 10–9. F_{MAX} Results from the Quartus II Timing Analysis with SignalTap II Logic Analyzer After Back-Annotation

Slack (ns)	Actual F_{MAX} (MHz)	From	To	Clock Source
0.053	125.83	state_m:inst1 filter~22	acc:inst3 result[11]	clk
0.196	128.14	taps:inst xn[0]~reg0	acc:inst3 result[11]	clk
0.171	127.89	state_m:inst1 filter~22	acc:inst3 result[6]	clk
0.171	127.89	state_m:inst1 filter~22	acc:inst3 result[7]	clk

By back-annotating your original design, the register-to-register delay decreases significantly and the original timing requirements are met.

Figure 10–25 shows the timing closure floorplan editor.

Figure 10–25. Timing Closure Floorplan Editor



Design Example: Using SignalTap II Logic Analyzers in SOPC Builder Systems

Application Note 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, an 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 on this example, see *Application Note 323: Using SignalTap II Embedded Logic Analyzers in SOPC Builder Systems*.

Conclusion

As the FPGA industry continues to make technological advancements, outdated methodologies need to be replaced with a new set of 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 SignalTap II Logic Analyzer provides many new and

innovative features, allowing you to capture and analyze internal signals in your FPGA, thereby allowing you to find the source of a design flaw in the shortest amount of time.

Introduction

One of the toughest challenges that FPGA designers face is implementing engineering change orders (ECOs) late in the design cycle. When altering functionality late in the design cycle, issues such as preserving functionality and timing while meeting the change request specifications in the shortest amount of time are critical. With the Quartus® II software's Chip Editor, you can view the internal structure of Altera® devices and edit functionality and parameter settings for resources within the device. The Chip Editor also helps you document and manage all of the ECOs in your design.

The Chip Editor works directly on the post place-and-route design database so you can implement device changes in minutes without performing a full compilation. The changes you make are restricted to particular device resources. Therefore, the timing performance of the remaining portions of your design are not affected. Design rule checks are performed on all changes to prevent you from making illegal modifications to your design.

This chapter describes how to use the Chip Editor and includes coverage of the following topics:

- Chip editor floorplan
- Resource property editor
- Common applications

Background

With the Chip Editor, you can view the following architecture-specific information related to your design:

- Device routing resources used by your design: For example, you can visually examine how blocks are connected, as well as the signal routing that connects the blocks.
- LE configuration: You can 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: You can 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: You can view how the device I/O resources are used. For example, you can view what 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: You can 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 along with the settings for your PLL.

With the Chip Editor, you can modify:

- LEs and ALMs
- I/O cells
- PLLs
- Connections between elements
- Placement of elements

The Chip Editor supports the following Altera device families:

- Stratix® II
- Stratix
- Stratix GX
- Cyclone™
- MAX® II

Using the Chip Editor in Your Design Flow

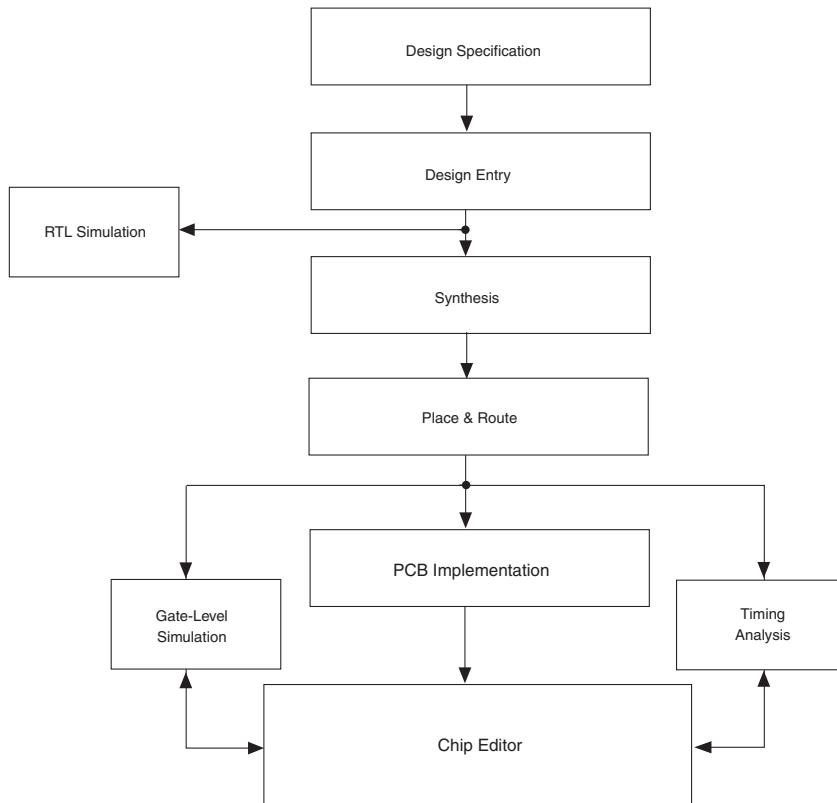
The ideal design flow starts by developing the design specification, creating register transfer level (RTL) code that describes the design specification, verifying that the RTL code performs the correct functionality, verifying that the fitted design satisfies the design's timing constraints, and ends with successfully programming the targeted FPGA.

Unfortunately, similar to most difficult processes, the ideal design flow rarely occurs. Often, you find bugs in the RTL code or the design specifications change midway through the design cycle. The challenge lies in efficiently accommodating these types of design issues. Traditionally, you have to go back to the source RTL code, make the appropriate changes, and then go through the entire design flow again, including synthesis, place-and-route, and verification.

With the Altera Chip Editor, you can shorten the design cycle time. When changes are made to your design, you do not have to perform a full compilation in the Quartus II software. You make changes directly to the post place-and-route netlist, generate a new programming file, and test the revised design by performing a gate-level timing simulation and timing analysis without modifying the RTL code. You can continue to

iteratively make changes to your design using the Chip Editor until you correct your problem. [Figure 11–1](#) describes how you can use the Chip Editor in your design flow.

Figure 11–1. Chip Editor Design Flow



Chip Editor Features

The Chip Editor contains many advanced features that enable you to quickly and efficiently make design changes. The Chip Editor's integrated tool set provides the following features:

- Chip Editor Floorplan: Allows you to view post-compilation placement, connections, and routing paths; create new logic cells and I/O atoms and move existing logic cells and I/O atoms while having full view of your design
- Resource Property Editor: Allows you to make changes to the properties and parameters of resources, and modify connectivity between certain types of resources
- Change Manager: Maintains an ongoing record of your post-compilation changes, and assists in ensuring that conflicting changes do not occur; you can also write out a tool command language (.tcl) file that contains a list of all of your changes

The Chip Editor allows you to quickly and easily view post-compilation placement and routing information. You can start the Chip Editor in any of the following ways:

- Choose **Chip Editor** (Tools menu)
- Right button pop-up menu from the Compilation Report
- Right button pop-up menu from the RTL source code
- Right button pop-up menu from the Timing Closure Floorplan
- Right button pop-up menu from the Node Finder
- Right button pop-up menu from the Simulation Report
- Right button pop-up menu from the RTL Viewer

Chip Editor Floorplan

Table 11–1 gives a summary of the Chip Editor Floorplan features. These features are accessible from the Chip Editor Toolbar.

Table 11–1. Chip Editor Floorplan Features (Part 1 of 2)

Feature	Description
Fan-In Connections	Generates the connections to the selected resource
Fan-Out Connections	Generates the connections away from the selected resource
Immediate Fan-In	Generates the routing fan-in connections for the selected resource
Immediate Fan-Out	Generates the routing fan-out connections for the selected resources
Show Delays	Displays the time delay between the two selected resources

Table 11–1. Chip Editor Floorplan Features (Part 2 of 2)

Feature	Description
Generate Connections Between Nodes	Displays the routing connections between selected nodes
Expand Connections	Displays sub-connections between two nodes
Bird's Eye View	Gives a high-level picture of resource usage of the whole chip, allows you to specify which elements of the Chip Editor Floorplan are displayed, and assists you in rapidly navigating the floorplan



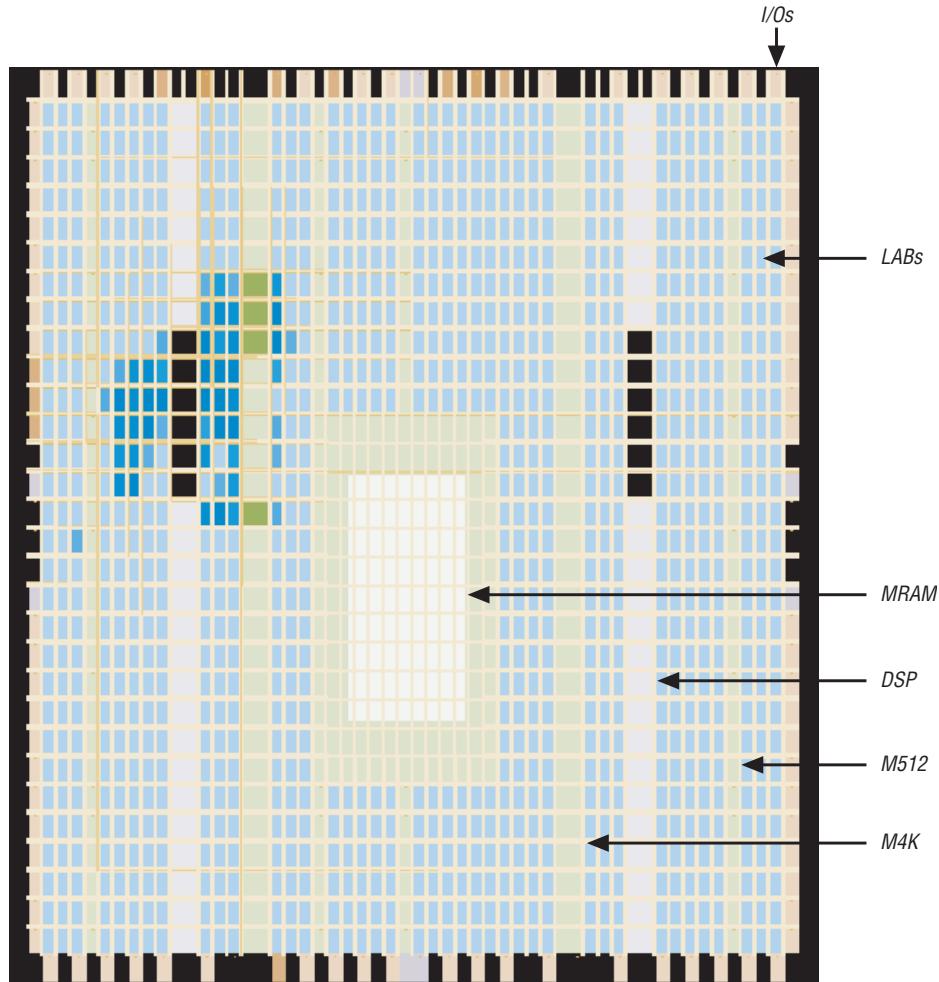
For a detailed description of all the elements in the Chip Editor toolbar, refer to the Quartus II help.

The Chip Editor Floorplan 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 (Highest) Level View

The first (highest) 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 11–2](#) shows the Chip Editor’s first level view.

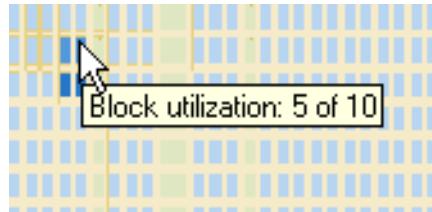
Figure 11–2. Chip Editor's First (Highest) Level View



Each resource is shown in a different color, making it easier to distinguish between resources. The Chip Editor 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 (see Figure 11–3).

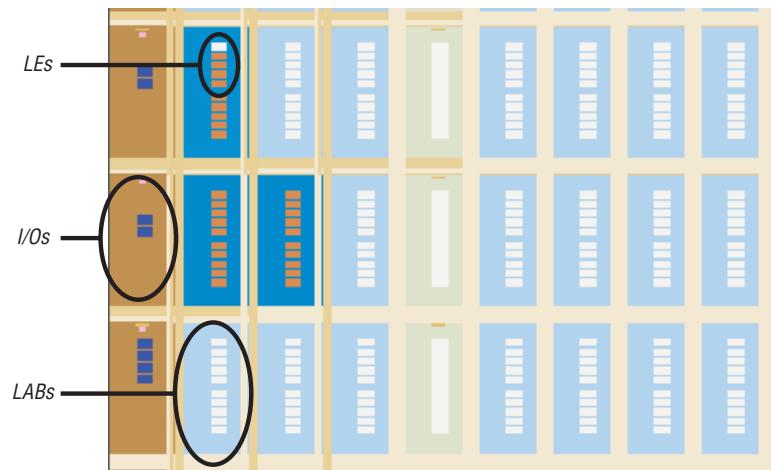
Figure 11–3. Tooltip Message: First Level View



Second Level View

As you zoom in, you see an increase in the level of detail. Figure 11–4 shows the Chip Editor's second level view.

Figure 11–4. Chip Editor's Second Level View



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 together.

When you place the mouse pointer over an LE at this level, a tooltip is displayed that shows the name of the LE, the location of the LE, 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 11–5 shows the level 2 tooltip information.

Figure 11–5. Tooltip Message: Second Level View

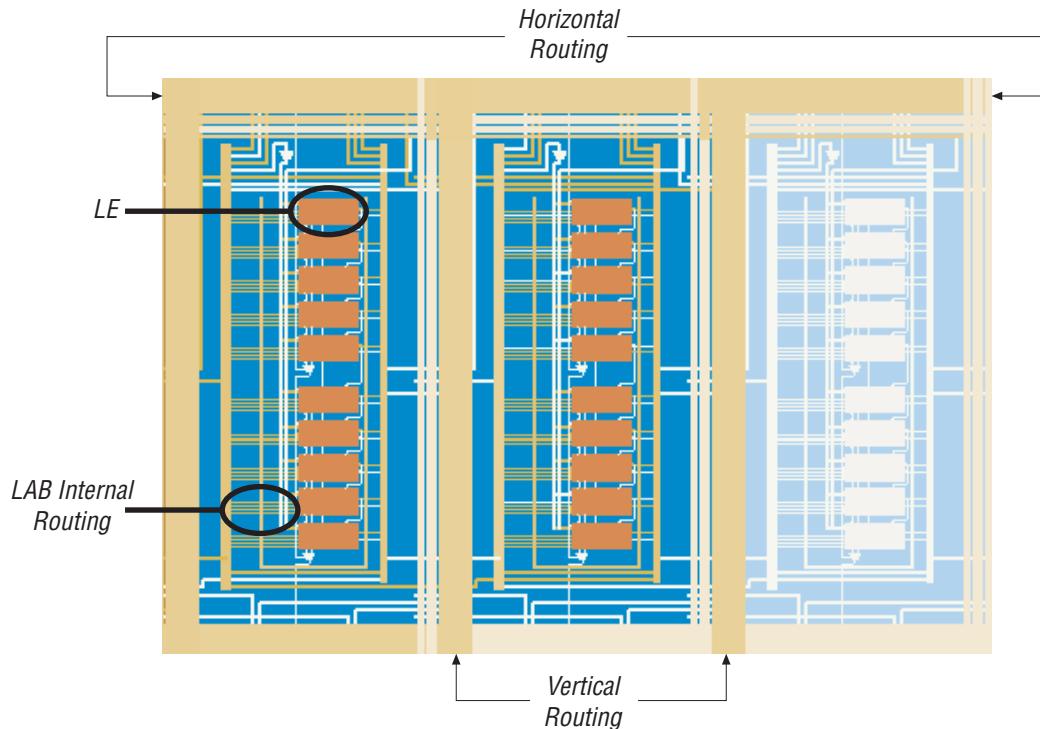


Third Level View

Figure 11–6 shows the level of detail at the third and lowest level. At this level you can see each routing resource that is used within a LAB in the FPGA.

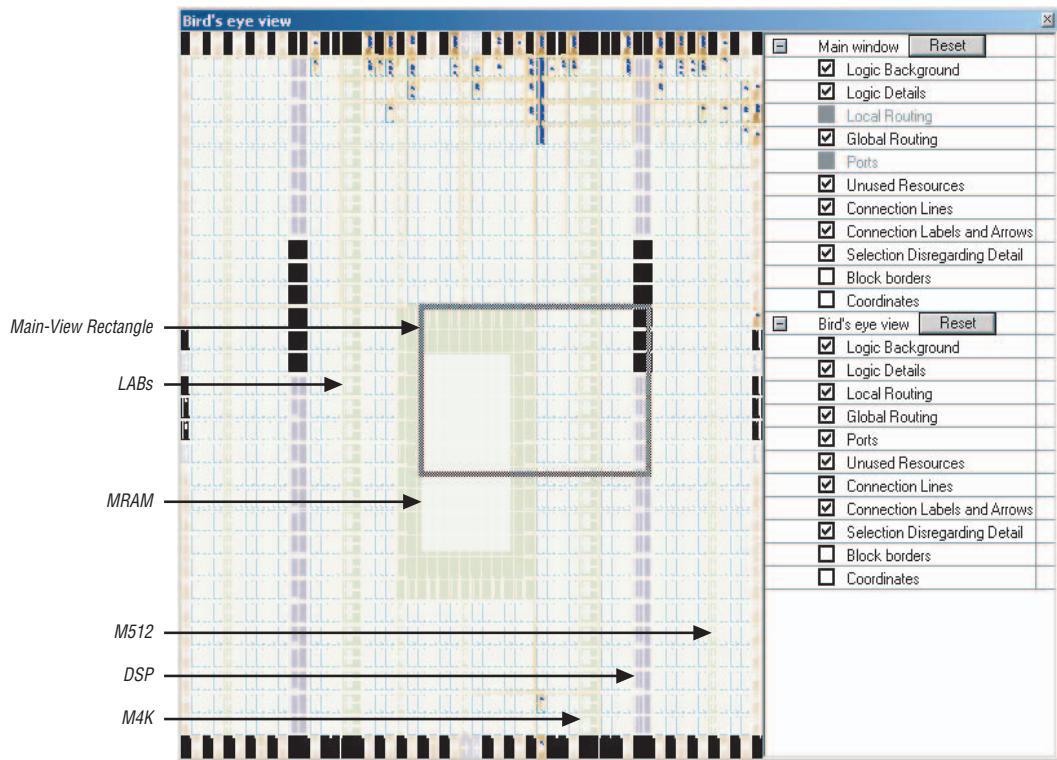
At this level, you can move LEs and I/Os from one physical location to another. You can move a resource by selecting, dragging, and dropping it into the desired location. At this level you also can create new LEs and I/Os. To create a resource, right click at the location you want to create the resource and select **Create Atom** (right button pop-up menu). To delete a resource, right click on the resources you want to delete and select **Delete Atom**. You can only delete a resource after all of its fan-out connections are removed.

Figure 11–6. Chip Editor’s Third Level View



Bird’s Eye View

The Bird’s Eye View (see Figure 11–7) displays a high-level picture of resource usage for the entire chip. It provides a fast and efficient means of navigating between areas of interest in the Chip Editor. In addition, it provides controls that allow you to specify which graphic elements are displayed. The specifications can be applied either to the Bird’s Eye View and the main Chip Editor window.

Figure 11–7. Bird's Eye View

The Bird's Eye View is displayed as a separate window that is linked to the Chip Editor. When you select an area of interest in the Bird's Eye View, the Chip Editor 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 Editor 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 Editor 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.

Resource Property Editor

You can view and edit the following elements with the Resource Property Editor:

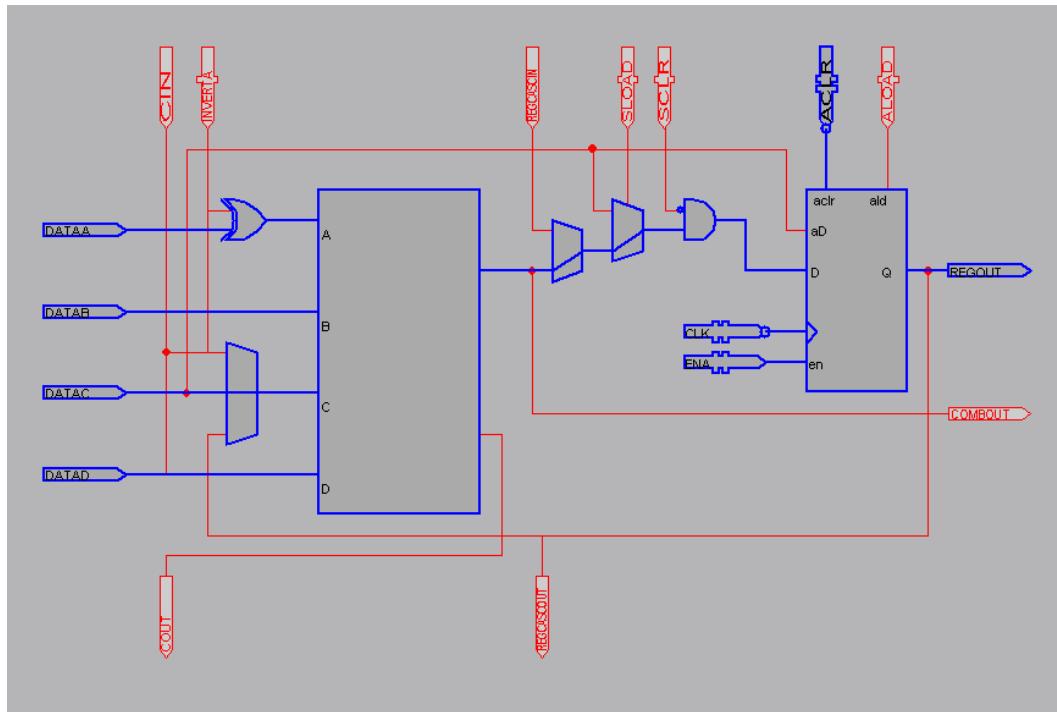
- LEs (Stratix, Stratix GX, Cyclone, MAX II)
- ALMs (Stratix II)
- I/O resources
- PLLs

LE Properties

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 can be fed by the output of the LUT or by an independent function generated in a separate LE. [Figure 11–8](#) shows what the LE looks like in the Resource Property Editor. [Figure 11–8](#) shows an LE that uses the DATAAC and DATAD inputs and the COMBOUT output.

You can use the Resource Property Editor to view and edit any LE that is placed in the FPGA. Right click on an LE in the Timing Closure, RTL Viewer, Node Finder, or Chip Editor and select **Locate in Resource Property Editor** from the right button pop-up menu to open the Resource Property Editor for an LE.

For more information on LE architecture for a particular device family, refer to the device family handbook or data sheet.

Figure 11–8. Stratix LE Architecture Note (1), (2)**Note to Figure 11–8:**

- (1) By default, the Quartus II software displays the used resources in blue and the unused in grey. For this figure, the used resources are in blue and the unused resources are in red.
- (2) For more information on the Stratix device's LE Architecture, refer to the *Stratix Device Handbook*.

This section discusses the following properties of the LE that you can examine using the Resource Property Editor:

- Mode of operation (normal or arithmetic)
- LUT equation
- LUT mask
- Synchronous mode
- Register cascade mode

Supported Changes for an LE

You can use the Resource Property Editor to view, change connectivity, and edit the properties of the LEs. You can use the Chip Editor Floorplan to change placement, delete, and create new LEs. You can perform these operations on Stratix, Stratix GX, Cyclone, and MAX II devices.

Mode of Operation

An LE can operate in either normal or arithmetic mode. For more information about the modes of operation, see Volume 1 of the *Stratix Device Handbook*, Volume 1 of the *Cyclone Device Handbook*, or the *MAX II Device Handbook*.

When an LE is configured in normal mode, the LUTs in the LE can implement a function of four inputs.

When the LE is configured in arithmetic mode, the LUTs in the LE are divided into 2 three-input LUTs. The first LUT generates the signal that drives the output of the LUT, while the second LUT is used to generate the carry-out signal. The carry-out signal can drive only a carry-in signal of another LE.

LUT Equation

You can change the logic function implemented by the LUT by changing the LUT equation. 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 equation.

The LUT mask is the hexadecimal representation of the LUT equation. When you change the LUT equation, the Quartus II software automatically changes the LUT mask.



For information on LUT mask, refer to “[LUT Mask](#)” on [page 11–16](#).

To change the function implemented by the LUT, you must first understand how the LUT works. A LUT contains storage cells that implement small logic blocks as a function of the inputs. Each storage cell is capable of holding a logic value, either a 0 or a 1. The Stratix, Stratix GX, Cyclone, and MAX II device families use a four-input LUT and have 16 storage cells. The LUT can store 16 output values in its storage cells. The output from the LUT depends on the signal that is driven into the input ports of the LUT.

Example of Computing the LUT Mask

Assume that you need to build the following logic function:

$$(A \text{ XOR } B) \text{ OR } (C \text{ AND } D)$$

Table 11–2 can be used to compute the LUT Mask for the equation above.

Table 11–2. LUT Mask Truth Table				
D Input	C Input	B Input	A Input	Output
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

The LUT mask is the hexadecimal representation of the LUT output. For example, the LUT output of (A XOR B) OR (C AND D) is represented by the following binary number: 1111011001100110. The LUT mask, in hexadecimal format for this binary number is F666.

When the LE is set to arithmetic mode, the first eight bits in the LUT mask represent the SUM equation output. The second eight bits represent the CARRY equation.

When a change is made to the LUT mask, the Quartus II software automatically computes the LUT equation.

Synchronous Mode

When an LE is in synchronous mode, the synchronous load (`sload`) and synchronous clear (`sclr`) signals are used. You can change the synchronous mode of an LE by connecting (or disconnecting) the `sload` and `sclr`.

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 must be the same for all other LEs in the same LAB. This includes the inversion state of the signal. 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 a series of shift registers. You can change the register cascade mode by connecting (or disconnecting) the cascade-in. However, if you are creating this port, you must ensure that the source LE is directly above the destination LE.

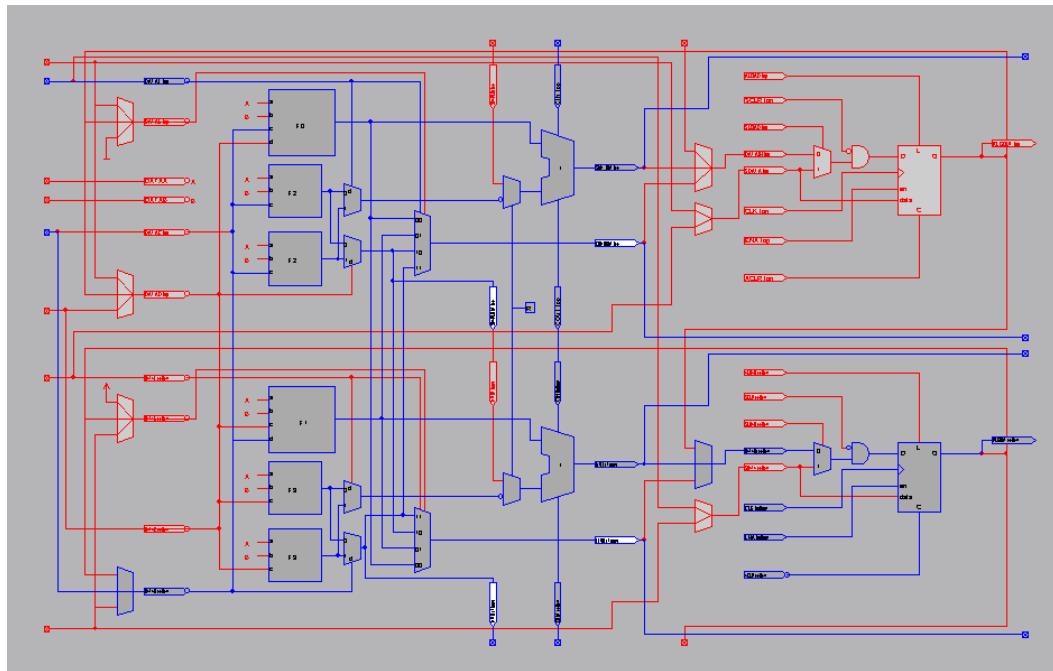
ALM Properties

The basic building block of logic in the Stratix II architecture is the ALM (see [Figure 11-9](#)). The ALM provides advanced features with efficient logic utilization. Each ALM contains a variety of 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. Through these dedicated resources, the ALM can efficiently implement various arithmetic functions and shift registers.

You can view and edit any ALM in a Stratix II device with the Resource Property Editor. To view a specific ALM in the Resource Property Editor, right-click on the ALM in the Timing Closure Floorplan, RTL Viewer, Node Finder, or Chip Editor, and select **Locate in Resource Property Editor** from the right button pop-up menu.



For a detailed description of the ALM, refer to the *Stratix II Device Handbook*.

Figure 11–9. ALM Architecture Note (1)**Note to Figure 11–9:**

- (1) By default, the Quartus II software displays the used resources in blue and the unused in grey. For this figure, the used resources are in blue and the unused resources are in red.

Supported Changes for an ALM

You can use the Resource Property Editor to view, change connectivity, and edit the properties of the ALMs. You can use the Chip Editor Floorplan to change placement, delete, and create new ALMs. These operations can be performed on Stratix, Stratix GX, Cyclone, and MAX II devices.

LUT Mask

The LUT mask is the hexadecimal representation of the LUT output. Each ALM is broken down into a ‘top’ LUT and a ‘bottom’ LUT. The LUT mask for each LUT is computed in the same manner as is shown in the “[Example of Computing the LUT Mask](#)” on page 11–13. However, instead of four inputs, six inputs are used. Since the LUTs are driven by six inputs, the LUT output is represented by a 64-bit binary number or a 16-digit hexadecimal number.

The following examples illustrate the use of the LUT mask of an ALM.

Example 1:

If the ALM implements a logical AND function in the ‘top’ LUT using the DATAE and DATAF inputs, you will get the following:

LUT Mask: 000000000000FFFF

COMBOUT Equation: DATAE & DATAF

Example 2:

If the ALM implements a logical XOR function in the ‘top’ LUT using the DATAE and DATAF inputs, the LUT mask and COMBOUT equation are as shown below:

LUT Mask: 0000FFFFFFF0000

COMBOUT Equation: (!DATAE & DATAF) # (!DATAF & DATAE)

Extended LUT Mode

When using the extended LUT mode, the ALM creates a specific set of seven-input functions. The Quartus II software recognizes the applicable seven-input function automatically and fits them into an ALM. The ‘top’ and ‘bottom’ LUTs are each functions of five inputs, in which four of the inputs are shared. The other input of the ALM controls the multiplexer (MUX), which selects which LUT is used to drive the COMBOUT port.



For more information on Extended Mode, refer to the *Stratix II Device Handbook*.

Shared Arithmetic Mode

When an ALM is in arithmetic mode, it uses two sets of two four-input LUTs along with two dedicated full adders. The carry-in signal that feeds the ALM drives adder0. The carry-out from adder0 feeds the carry-in of adder1. The carry-out from adder1 drives adder0 of the next ALM in the LAB. ALMs in arithmetic mode can drive out registered and/or unregistered versions of the adder outputs.



For more information on Shared Arithmetics Mode refer to the *Stratix II Device Handbook*.

FPGA I/O Elements

Altera FPGAs, with high-performance I/O elements packed with up to six registers, are equipped with support for a number of I/O element standards allowing you to run your design at peak speeds.



For a detailed description of the Stratix device I/O element, see the *Stratix Architecture* chapter in Volume 1 of the *Stratix Device Handbook*.

For a detailed description of the Stratix II device I/O element, see the *Stratix II Architecture* chapter in Volume 1 of the *Stratix II Device Handbook*.

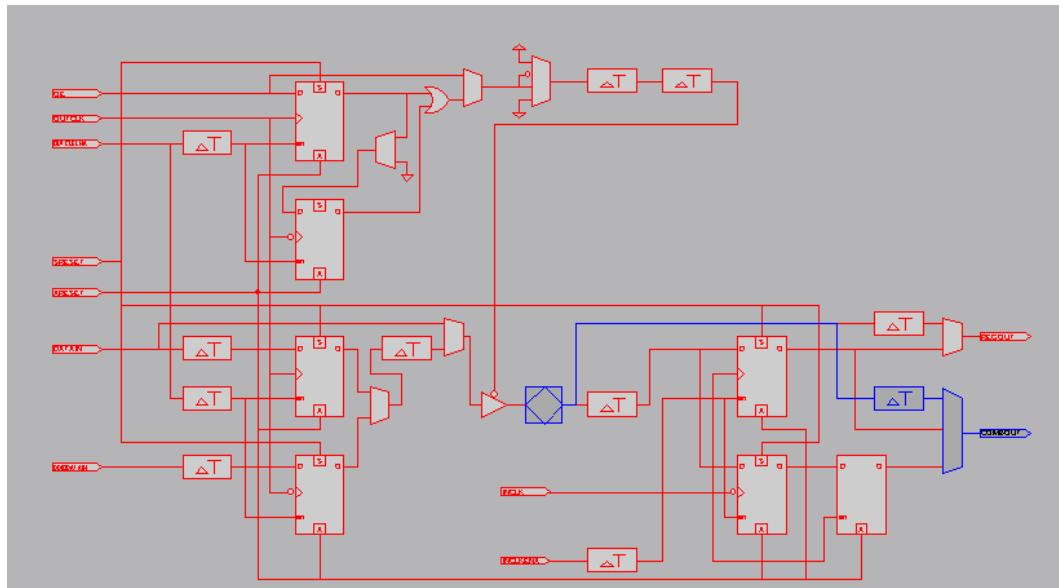
For a detailed description of the Cyclone device I/O element, see the *Cyclone Architecture* chapter in Volume 1 of the *Cyclone Device Handbook*.

For a detailed description of the MAX II device I/O element, see the *MAX II Architecture* chapter in the *MAX II Device Handbook*.

Stratix, Stratix GX, & Stratix II I/O Elements

The I/O element in Stratix devices contains a bidirectional I/O buffer, six registers, and a latch for a complete bidirectional single data rate or DDR transfer. [Figure 11-10](#) shows the Stratix and Stratix GX I/O element structure. The I/O element contains two input registers (plus a latch), two output registers, and two output enable registers.

Figure 11–10. Stratix & Stratix GX Device I/O Element Note (1), (2)

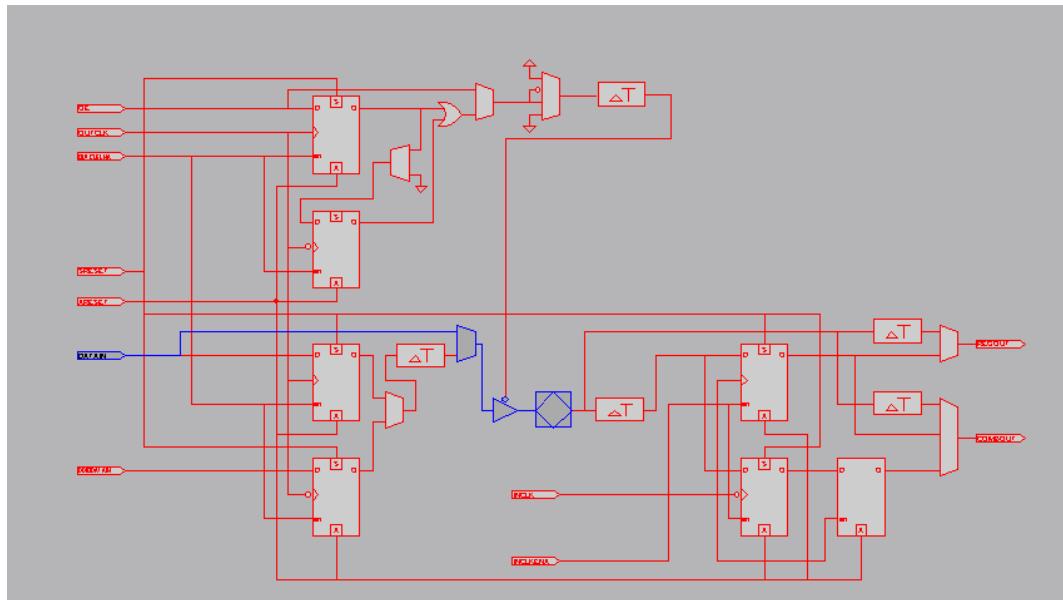


Note to Figure 11–10:

- (1) By default, the Quartus II software displays the used resources in blue and the unused in grey. For this figure, the used resources are in blue and the unused resources are in red.
- (2) For more information on I/O elements in Stratix and Stratix GX devices, refer to the *Stratix Device Handbook* and the *Stratix GX Handbook*.

Figure 11–11 shows the Stratix II I/O element structure.

Figure 11–11. Stratix II Device I/O Element Note (1), (2)



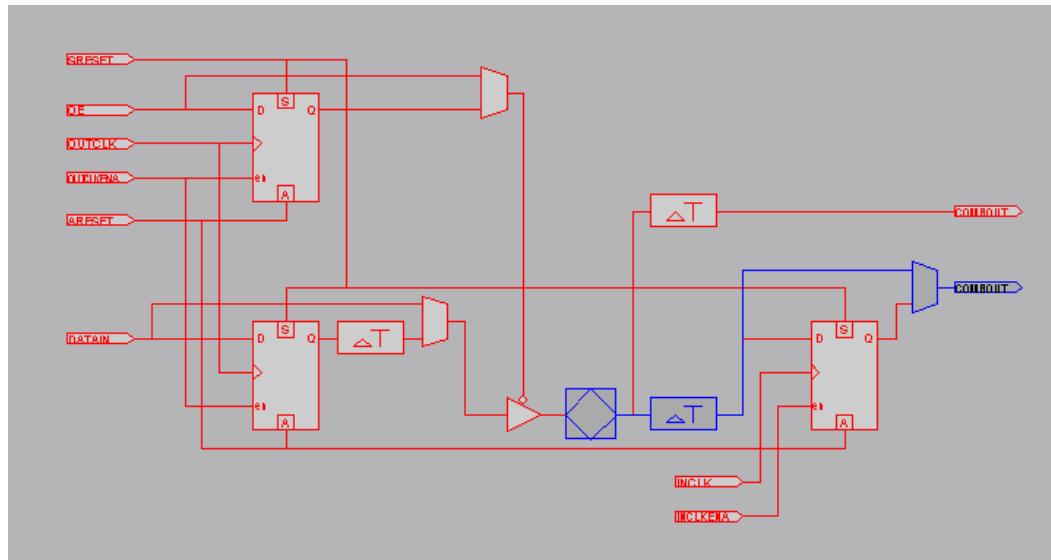
Note to Figure 11–11:

- (1) By default, the Quartus II software displays the used resources in blue and the unused in grey. For this figure, the used resources are in blue and the unused resources are in red.
- (2) For more information on I/O elements in Stratix II devices, refer to the *Stratix II Device Handbook*.

Cyclone I/O Elements

The I/O elements in Cyclone device contain a bidirectional I/O buffer and three registers for complete bidirectional single data rate transfer. Figure 11–12 shows the Cyclone I/O element structure. The I/O element contains one input register, one output register, and one output enable register.

Figure 11–12. Cyclone Device I/O Element Note (1), (2)



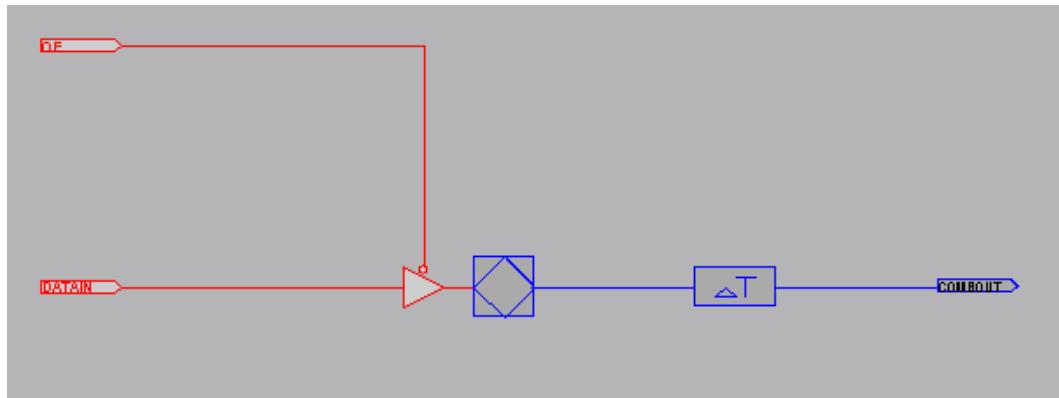
Note to Figure 11–12:

- (1) By default, the Quartus II software displays the used resources in blue and the unused in grey. For this figure, the used resources are in blue and the unused resources are in red.
- (2) For more information on I/O elements in Cyclone devices, refer to the *Cyclone Device Handbook*.

MAX II I/O Elements

MAX II device I/O elements contain a bidirectional I/O buffer.

Figure 11–13 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 11–13. MAX II Device I/O Element Note (1), (2)**Note for Figure 11–13:**

- (1) By default, the Quartus II software displays the used resources in blue and the unused resources in grey. For this figure, the used resources are in blue and the unused resources are in red.
- (2) For more information on I/O elements in MAX II devices, refer to the *MAX II Device Handbook*.

I/O Elements Features in the Resource Property Editor

- You can use the Resource Property Editor to view, change connectivity, and edit the properties of the I/O elements. You can use the Chip Editor Floorplan to change placement, delete, and create new I/O elements. These operations can be performed on Stratix, Stratix GX, Cyclone, and MAX II devices.

Table 11–3 describes the I/O properties that are editable using the Resource Property Editor.

Table 11–3. Editable I/O Properties Using the Resource Property Editor (Part 1 of 2)				
I/O Feature	Stratix II	Stratix & Stratix GX	Cyclone	MAX II
Bus Hold	✓	✓	✓	✓
Weak Pull Up	✓	✓	✓	✓
Slow Slew Rate	✓	✓	✓	✓
Open Drain	✓	✓	✓	✓
I/O Standard	✓	✓	✓	✓
Current Strength	✓	✓	✓	✓
Extend OE Disable	✓	✓	✓	✓

Table 11–3. Editable I/O Properties Using the Resource Property Editor (Part 2 of 2)

I/O Feature	Stratix II	Stratix & Stratix GX	Cyclone	MAX II
PCI I/O	✓	✓	✓	✓
On-Chip Termination	✓	✓	✓	
Input Register Reset Mode	✓	✓	✓	
Input Register Synchronous Reset Mode	✓	✓	✓	
Input Powers Up	✓	✓	✓	
Output Register Mode	✓	✓	✓	
Output Register Reset Mode	✓	✓	✓	
Output Register Synchronous Reset Mode	✓	✓	✓	
Output Powers Up	✓	✓	✓	
OE Register Mode	✓	✓	✓	
OE Register Reset Mode	✓	✓	✓	
OE Register Synchronous Reset Mode	✓	✓	✓	
OE Powers Up	✓	✓	✓	
Input Clock Enable Delay	✓	✓		
Output Clock Enable Delay	✓	✓		
Output Enable Clock Enable Delay		✓		
Input Pin to Logic Array Delay	✓	✓	✓	✓
Output Pin Delay	✓	✓	✓	
Input Pin to Input Register Delay	✓	✓	✓	
Output Enable Register tCO Delay	✓	✓		
Output tZX Delay	✓	✓		
Logic Array to Output Register Delay	✓	✓		

Modifying the PLL Using the Chip Editor

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.

PLL Properties

You can change many of the PLL properties with the Resource Property Editor. You can modify the following internal parameters of the PLL:

The in-loop parameters that can be modified include:

- M initial
- M
- Counter Time Delay M
- M Tap VCO
- N
- Counter Time Delay N
- M2
- N2
- Loop filter resistance
- Loop filter capacitance
- Charge pump current

The post-loop parameters that can be modified include:

- Counter high
- Counter low
- Counter PH
- Counter initial
- Counter time delay

Adjusting the Duty Cycle

Use the following equations to adjust the duty cycle of individual output clocks:

$$\text{High \%} = \text{Counter High}/(\text{Counter High} + \text{Counter Low})$$

$$\text{Low \%} = \text{Counter Low}/(\text{Counter High} + \text{Counter Low})$$

Adjusting the Phase Shift

Use the following equations to adjust the phase shift of an output clock of a PLL:

$$\text{Phase Shift} = (\text{Period VCO} \times 0.125 \times \text{Tap VCO}) + (\text{Initial VCO} \times \text{Period VCO})$$



For a detailed description of the settings, see the Quartus II Help. For more information on Stratix device PLLs, see the *Stratix Architecture* chapter in Volume 1 of the *Stratix Device Handbook*.

For normal node, calculate the phase shift with the following settings:

$$\text{Tap VCO} = \text{Counter PH} - \text{M Tap VCO}$$

$$\text{Initial VCO} = \text{Counter Initial} - \text{M Initial}$$

$$\text{Period VCO} = \text{In Clock Period} \times \text{N} / \text{M}$$

For external feedback mode, calculate the phase shift with the following settings:

$$\text{Tap VCO} = \text{Counter PH} - M \text{ Tap VCO}$$

$$\text{Initial VCO} = \text{Counter Initial} - M \text{ Initial}$$

$$\text{Period VCO} = \text{In Clock Period} \times N / (M + \text{Counter High} + \text{Counter Low})$$

Adjusting the Output Clock Frequency

Use the following equations to adjust the PLL output clock in normal mode:

$$\text{PLL numerator} = M + \text{counter high} + \text{counter low}$$

$$\text{PLL denominator} = N(\text{counter high} + \text{counter low})$$

Use the following equation to adjust the PLL output clock in external feedback mode:

$$\text{OUTCLK} = \text{INCLK} \cdot \frac{M + \text{counter high} + \text{counter low}}{N + \text{counter high} + \text{counter low}}$$

You can adjust all the output clocks by modifying the M and N values. You can adjust individual output locks by modifying the counter high and counter low values.

Adjusting the Spread Spectrum

Use the following equation to adjust the spread spectrum for your PLL:

$$\% \text{spread} = 1 - \frac{M_2 N_1}{M_1 N_2}$$

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 change that you have made with the Resource Property Editor. Each change is numbered sequentially, such that the larger the number the more recent the change.

More complex changes are marked in the Change Manager with a “+”. You can expand a complex entry in the Change Manager to reveal all the changes that have occurred. An example of a complex change would be creation or deletion of an atom.

Table 11–4 summarizes the information shown by the Change Manager.

Table 11–4. Change Manager Information	
Column Name	Description
Node name	Name of the node modified with the Chip Editor
Change type	Type of change made to the node
Old value	Value previous to the change
Target value	Value of the change that you want to set (before a Check and Save has been performed)
Current value	Value in the currently viewed netlist. This value is not necessarily ready for POF generation
Disk value	Current value of the node as contained within the assembler netlist (Value available for use in the Assembler, Timing Analysis, Simulation)
Comments	User comments

Once you have completed all your design modifications, you need must check the integrity of the netlist after the applied changes by right clicking in the **Change Manager** and selecting **Check & Save All Netlist Changes** (right button pop-up menu). If the applied changes successfully pass the netlist check, they are written to disk. If all applied changes do not pass the netlist check, all changes made since the previous successful netlist check are reversed. Figure 11–14 shows the Change Manager.

The colors in the Current Value and Disk Value columns indicate the present status of the data in those columns. When an entry is green in the Current Value column, the change has occurred and the value is the same as the value in memory. When an entry is blue in the Disk Value column, the change has successfully passed a **Check & Save Netlist** command, and the value in that column is the same as the value on disk.

Figure 11–14. Change Manager Results

Index	Node Name	Change Type	Old Value	Target Value	Current Value	Disk Value	Comment
1	lchiptriplock_cnt:tick bcount:counter p0count:sub 6	Location Index	LC_X46_Y23_N1	LC_X46_Y23_N0	LC_X46_Y23_N0	LC_X46_Y23_N0	
2	lchiptripauto_max:auto gdf~177	Location Index	LC_X46_Y23_N2	LC_X45_Y23_N2	LC_X45_Y23_N0	LC_X45_Y23_N2	
⊕ 3	New_atom_1	New Lcell	None	Exists	Exists	None	
4	lchiptripauto_max:auto gdf~179	Location Index	LC_X46_Y23_N3	LC_X46_Y23_N2	LC_X46_Y23_N2	LC_X46_Y23_N3	
⊕ 5	lchiptriplock_cnt:tick bcount:counter p0count:su...	Modify Source	Disconnected	VCC	VCC	Disconnected	
⊕ 6	lchiptripauto_max:auto gdf~179:REGOUT:0	Output Port Modific...	None	Exists	Exists	None	
7	lchiptripauto_max:auto gdf~177	Location Index	LC_X45_Y23_N2	LC_X45_Y23_N0	LC_X45_Y23_N0	LC_X45_Y23_N2	



Each line in the Change Manager represents a change record. Simple changes appear as a single line in Change Manager. More complex changes, which require that several actions be performed to achieve the change, appear as a single line marked by a “+” which you can click 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 Editor Floorplan** (including creating or deleting atoms and changing connectivity), may appear to be self-contained, however, these changes are actually composed of multiple actions. These types of changes are recorded with a “+” sign in the **Index** column. [Figure 11–15](#) shows the Change Manager.

[Figure 11–15. Change Manager](#)

Index	Node Name	Change Type	Old Value	Target Value	Current Value	Disk Value	Comment
+ 1	New_atom	New Lcell	None	Exists	Exists	None	

In the example shown in [Figure 11–15](#), a new ATOM is created. The change record in the **Change Manager** is a single-line representation of the actual change actions that have occurred. You can expand the change record to show the component actions that make up the change, by clicking the “+” icon as shown in [Figure 11–16](#) below.

[Figure 11–16. Change Manager Results](#)

Index	Node Name	Change Type	Old Value	Target Value	Current Value	Disk Value	Comment
+ 1	New_atom	New Lcell	None	Exists	Exists	None	
+ 1a	New_atom	New Lcell	None	Exists	Exists	None	
+ 1b	New_atom:COMBOUT:0	Output Port Modification	None	Exists	Exists	None	
+ 1c	[chiptrip]New_atom	Location Index	N/A	LC_X33_Y30_N5	LC_X33_Y30_N5	Data Not Available	

After clicking the “+” sign you can see that creation of an ATOM consists of three autonomous 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.

Common Applications

You can use the following Chip Editor functions to help build your system as quickly as possible:

- Routing an internal signal to an output pin
- Adjust the phase shift of a PLL to meet I/O timing
- Correct a functional flaw in a design

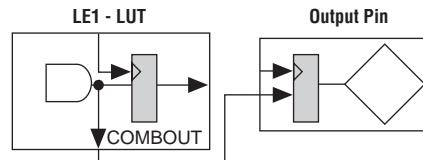
Routing an Internal Signal to an Output Pin

You can use the capabilities in the Chip Editor to route internal signals to unused output pins. This capability allows you to capture signals that are internal to the FPGA with an external logic analyzer.

The process of routing these signals is straightforward, and requires very little time, allowing you to spend less time on the setup and more time on debugging.

The following steps will help you understand the process required to route an internal signal to an output pin (see [Figure 11–17](#)).

Figure 11–17. Routing an Internal Signal to an Output Pin



1. Create an output pin.
2. If it doesn't already exist, create the REGOUT or COMBOUT of the source LE.
3. Connect the DATAIN of the output pin to the REGOUT or the COMBOUT of the source LE.
4. Optional—Connect a clock to the CLK port of the output pin. Connecting a clock allows you to register the signals before they are driven off-chip.

Adjust the Phase Shift of a PLL to Meet I/O Timing

Using a PLL in your design should help I/O timing. However, if your I/O timing requirements are still unmet, you can adjust the PLL phase shift to try to meet the I/O timing requirements of your design. Shifting the clock backwards will give a better t_{CO} at the expense of the t_{SU} , while shifting it forward will give a better t_{SU} at the expense of t_{CO} and t_H .

Use the equations shown in the PLL section to set the new phase shift value to optimize your I/O Timing.

Correcting a Design Flaw

You may find functional flaws while you are debugging your design. Traditionally, these flaws (bugs) are corrected by modifying the RTL code, and going through the entire design flow again. This process can be very time-consuming, because the process of synthesis and place-and-route may take a significant amount of time. However, with the Chip Editor, you can make a change to your design without having to repeat the synthesis and place-and-route process.

To make a change with the Chip Editor, you can modify the LUT equation (or the LUT mask) of an LE with the Resource Property Editor.

Example Design: Meeting I/O Timing



Meeting the timing requirements of a design can be a difficult task. There are a number of proven methods that you can use to correct timing issues; however, the most efficient method will vary depending on a number of factors. The following example demonstrates how using the Chip Editor can help you to meet the timing requirements in a design.

To download the design files, go to the *Quartus II Handbook* section of the Altera web site and find the **retiming.zip** link, in Volume 3, Chapter 10.

Scenario: The t_{CO} requirement for a particular design is 7.0 ns. This requirement must be met to ensure that the output data is latched correctly before being sent to a receiving device.

Based on the Quartus II place-and-route results, the timing analysis data is shown in [Table 11–5](#).

Table 11–5. Timing Analysis Data					
Slack	Required t_{CO} (ns)	Actual t_{CO} (ns)	From	To	From CLK
-0.388 ns	7.000	7.388	outff_a~9	out	clk
-0.361 ns	7.000	7.361	outff_a~11	out	clk
-0.338 ns	7.000	7.338	outff_a~8	out	clk
-0.062 ns	7.000	7.062	outff_a~10	out	clk

The equation for t_{CO} is defined as:

$$t_{CO} = \langle \text{clock to source register delay} \rangle + \langle \text{micro clock to output delay} \rangle + \langle \text{register to pin delay} \rangle$$

To meet the t_{CO} requirement, either the $\langle \text{clock-to-source register delay} \rangle$ or the $\langle \text{register-to-pin delay} \rangle$ (or both) need to be reduced.

Solution: Use the Chip Editor to manually perform gate-level retiming to correct the t_{CO} .

If we examine one of the four failing paths in the timing analysis report, we see the following results:

```

Info: Slack time is -388 ps for clock "clk" between source register
"outff_a~9" and destination pin "out"
Info: + tco requirement for source register and destination pin is 7.000
      ns
Info: - tco from clock to output pin is 7.388 ns
Info: + Longest clock path from clock "clk" to source register is
      2.952 ns
Info: 1: + IC(0.000 ns) + CELL(0.828 ns) = 0.828 ns; Loc. = PIN_L3;
      Fanout = 100; CLK Node = 'clk'
Info: 2: + IC(1.582 ns) + CELL(0.542 ns) = 2.952 ns; Loc. =
      LC_X18_Y30_N8; Fanout = 1; REG Node = 'outff_a~9'
Info: Total cell delay = 1.370 ns ( 46.41 % )
Info: Total interconnect delay = 1.582 ns ( 53.59 % )
Info: + Micro clock to output delay of source is 0.156 ns
Info: + Longest register to pin delay is 4.280 ns
Info: 1: + IC(0.000 ns) + CELL(0.000 ns) = 0.000 ns; Loc. =
      LC_X18_Y30_N8; Fanout = 1; REG Node = 'outff_a~9'
Info: 2: + IC(0.506 ns) + CELL(0.280 ns) = 0.786 ns; Loc. =
      LC_X17_Y30_N6; Fanout = 1; COMB Node = 'xx[0]~190'

```

```
Info: 3: + IC(1.090 ns) + CELL(2.404 ns) = 4.280 ns; Loc. = PIN_E14;  
      Fanout = 0; PIN Node = 'out'  
Info: Total cell delay = 2.684 ns ( 62.71 % )  
Info: Total interconnect delay = 1.596 ns ( 37.29 % )
```

There are several methods that you can use to meet the t_{CO} requirement. However, further investigation shows that the most efficient method is to reduce the register-to-pin delay using gate-level retiming.

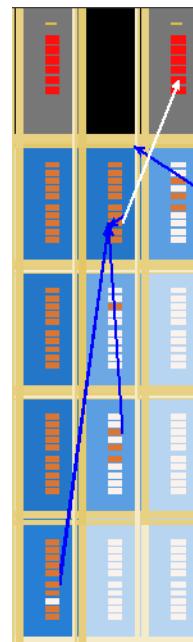
Based on the analysis just performed, you see that the data passes from the register, through the combinational logic, to the pin. You can move the register between the combinational logic and the pin to reduce the register-to-pin delay, thereby reducing the t_{CO} . It should be noted that by moving the register, the f_{MAX} of the overall circuit may decrease. Also, to use the manual gate-level retiming process you must ensure that moving the register does not alter the functionality of the circuit. In general, this method should be used only when you understand the design completely. If you are unsure about altering functionality, it is best to use the **Perform gate-level register retiming** option in the Quartus II software. Using this option in the Quartus II software requires a full compilation.

To reduce the register-to-pin delay, you need to move the register to the other side of the combinational logic. Perform this operation manually by following the steps shown below:

1. Locate the failing path in Chip Editor Floorplan (see [Figure 11-18](#)).

Right click in the t_{CO} section of the Timing Analysis Report (use the entry where the source register is outff_a~9) and select **Locate in Chip Editor** (right button pop-up menu).

Figure 11–18. Failing Path in Chip Editor

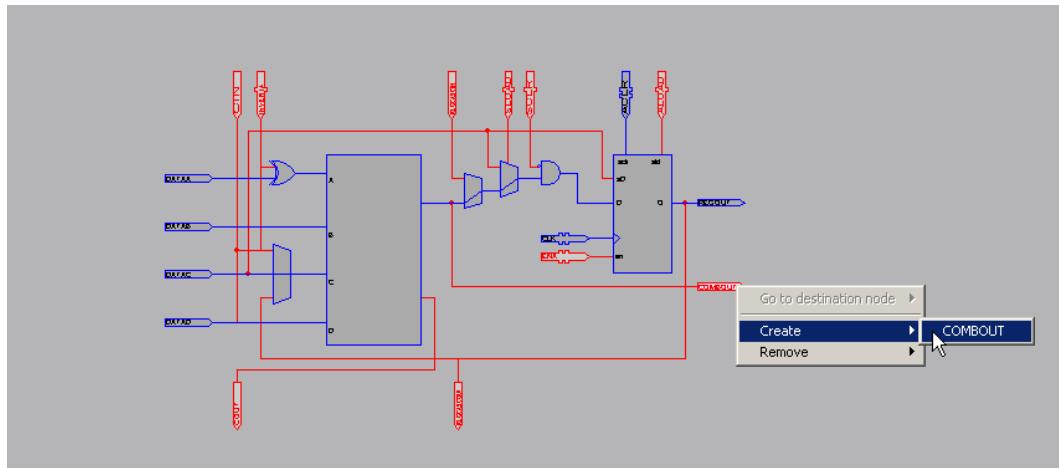


-
2. Open the Resource Property Editor and locate the source register.

Right click on the source register (outff_a~9) and select **Locate in Resource Property Editor** (right button pop-up menu).

3. Create the COMBOOUT port for outff_a~9.

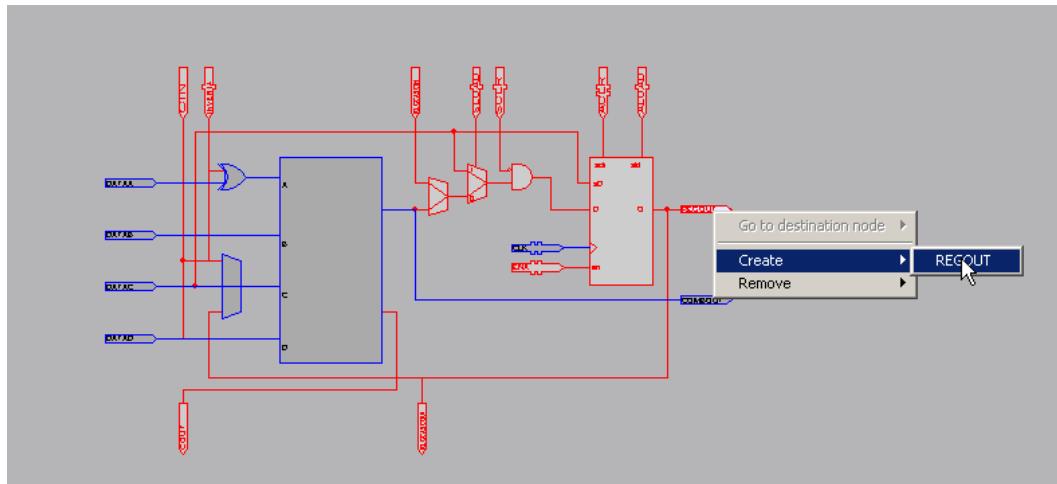
Right click the COMBOOUT port and select **Create COMBOOUT** (right button pop-up menu). See [Figure 11–19](#).

Figure 11–19. Select Create COMBOUT

4. Connect COMBOUT of outff_a~9 to DATA input of xx[0]~190.
 - a. Open the **Resource Property Editor** for xx[0]~190.
 - b. Right click on the DATA port of xx[0]~190 and select **Edit Connections>Other**. In the **Edit Connections** dialog box, find the outff_a~9~COMBOUT port with the **Node Finder**.

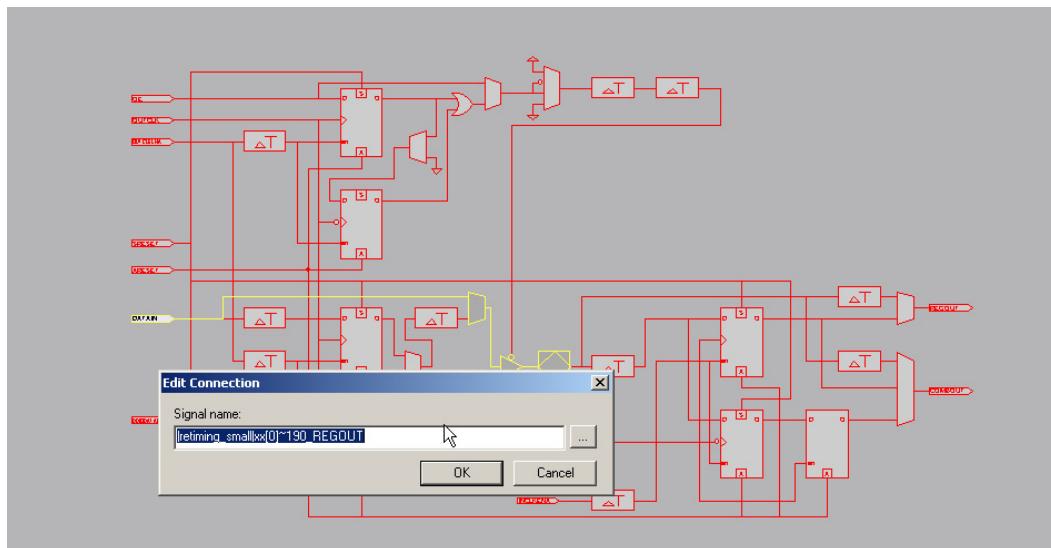
You have now removed the register from outff_a~9 and created a COMBOUT connection to xx[0]~190. The next step is to create the register in xx[0]~190.

5. Create the register in xx[0]~190.
 - a. Right click on the CLK port and select **Edit Connections**. In the **Edit Connections** dialog box, type clk (the name of the system clock in the design).
 - b. Right click on the REGOUT port and select **Create REGOUT** (see [Figure 11–20](#)).

Figure 11–20. Select Create REGOUT

6. Remove connection between COMBOUT of xx[0]~190 and DATAIN of out.
 - a. Right click on the COMBOUT of xx[0]~190 and select **Go To Destination atom out**.
 - b. Right click on the DATAIN port of out and select **Remove Connection**.
7. Connect REGOUT to DATAIN of the output pin-out.
 - a. Open the **Resource Property Editor** for out.
 - b. Right click on the DATAIN port of out and select **Edit Connections**. In the **Edit Connections** dialog box, find the REGOUT port for xx[0]~190 (use the **Node Finder**). See [Figure 11–21](#).

Figure 11–21. Select Edit Connections



8. Check and save netlist.

Right click in the **Change Manager** and select **Check & Save All Netlist Changes**.

You have now manually retimed your system to meet the t_{CO} requirements for one of the four failing paths. You must perform the same procedure on the other three paths to ensure the entire system meets the timing requirements. Once the other paths are fixed, you can run the Quartus II Timing Analyzer to verify the timing results and the Quartus II Simulator (or another EDA tool vendor's simulator) to verify the functionality of the design.

Table 11–6 describes the Timing Analysis report after the changes have been made.

Table 11–6. Timing Analysis Report After Changes Have Been Made					
Slack	Required t_{CO}	Actual t_{CO}	From	To	From CLK
0.405 ns	7.000 ns	6.595 ns	Outff_a~14	Out	Clk

Running the Quartus II Timing Analyzer

After you have made a change with the Chip Editor, you should perform timing analysis of your design with the Quartus II Timing Analyzer, to ensure that your changes have not adversely affected your design's timing performance.

For example, when you enable 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 need to perform timing analysis.

Once you make a change to your design using the Chip Editor, you should perform a timing simulation on your design with either the Quartus II Simulator or another EDA vendor's simulation tool.

Generating a Netlist for Other EDA Tools

When you use the Chip Editor, it may be necessary to verify the functionality using an Altera-supported simulation tool and/or verify timing using an Altera-supported timing analysis tool. You can run the Netlist Writer to generate a gate-level netlist that allows you to perform simulation or timing analysis in an EDA simulation or timing analysis tool of your choice.

Generating a Programming File

Once you have performed simulation and timing analysis, and are confident that the changes meet your design requirements, you can generate a programming file with the Quartus II Assembler. You use the programming file to implement your design in an Altera device.

Conclusion

As the time-to-market pressure mounts, it is increasingly important to be able to produce a fully-functional design in the shortest amount of time. To address this challenge, Altera developed the Quartus II Chip Editor. The Chip Editor enables you to modify the post place-and-route properties of your design. Specifically, you can change certain key properties of the LE, I/O elements, and PLL resources. Most importantly, changes made with the Chip Editor do not require a full recompilation, eliminating the lengthy process of RTL modification, resynthesis, and another place-and-route cycle.

In summary, the new features in the Chip Editor allow you to perform gate-level register retiming to optimize the timing of your design. The overall effect of using the Chip Editor shortens the verification cycle and brings timing closure to your design in a shorter period of time.



Section V. 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 chapter:

- [Chapter 12, In-System Updating of Memory & Constants](#)
- [Chapter 13, Cadence Incisive Conformal Equivalency Checker Support](#)
- [Chapter 14, Synopsys Formality Support](#)

Revision History

The table below shows the revision history for [Chapter 12](#) and [Chapter 14](#).

Chapter(s)	Date / Version	Changes Made
12	Dec. 2004 v1.2	<ul style="list-style-type: none">● Chapter 12 was formerly Chapter 11.● Updated Table 12–2.● Corrected the Verilog code for the <code>lpm_hint</code> parameter.● Re-organized the “Making Changes” segment into the “Editing Data Displayed in the Hex Editor” and “Importing & 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.
13	Dec. 2004 v2.1	<ul style="list-style-type: none">● Chapter 13 was formerly Chapter 12.● Updates to tables and figures.● New functionality for Quartus II software 4.2.
	June 2004 v2.0	<ul style="list-style-type: none">● Updates to tables and figures.● New functionality for Quartus II software 4.1.● This chapter was formerly chapter 11 in the previous section.
	Feb. 2004 v1.0	Initial release.
14	Jan 2005 v1.0	Initial release.

qii53012-1.2

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.

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 12–1](#) lists the types of memory supported by the MegaWizard® Plug-In Manager and the In-System Memory Content Editor.

Table 12–1. 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 12–2](#) lists support for in-system updating of memory and constants for the Stratix® II, Stratix, Cyclone™ II, Cyclone, APEX™ II, APEX 20K, and Mercury™ device families.

Table 12–2. Supported Megafunctions

MegaFunction	Stratix/Stratix II			Cyclone/ Cyclone II	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 12–2:

- (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 12–1](#).

To enable your memory or constant to be modifiable, perform the following steps:

1. Choose **MegaWizard Plug-In Manager** (Tools menu).
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. Choose **Start Compilation** (Processing menu).

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>";
```

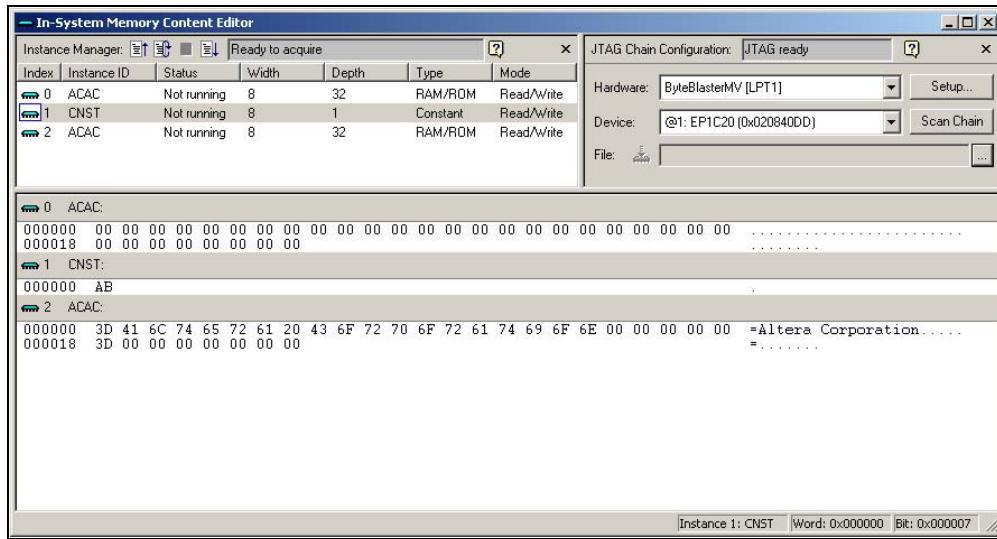
In Verilog HDL code, add the following:

```
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 (see Figure 12-1).

Figure 12–1. 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.

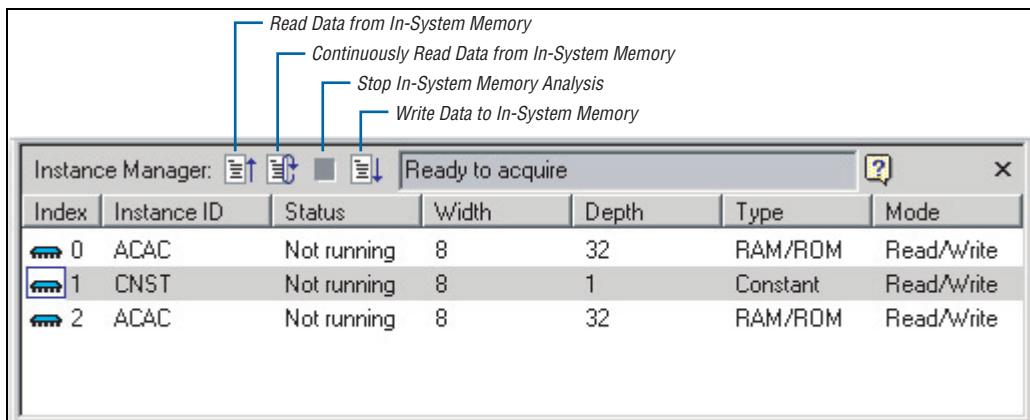
Each In-System Memory Content Editor can access the in-system memories and constants in a single device. If you have more than one device containing in-system configurable memories or constants in a JTAG chain, you can launch multiple In-System Memory Content Editors within the Quartus II software to 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 12–2](#).

Figure 12–2. 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

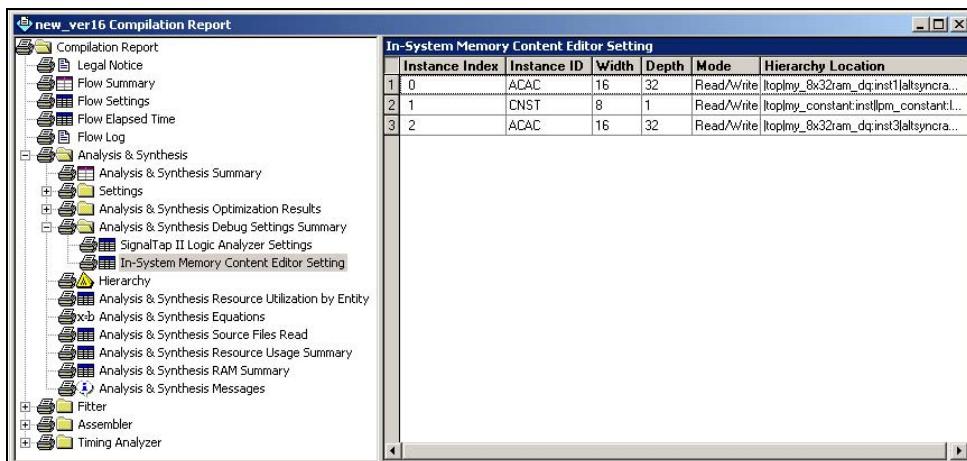


In addition to the buttons available in the Instance Manager, you can also read and write data by selecting the command from the Processing menu, or the right button pop-up menu in the Instance Manager or Hex Editor.

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 12–3](#)).

Figure 12–3. 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. You can also update a block of data by selecting a block of data and choosing either **Fill with 0's**, **Fill with 1's**, **Fill with Random Values**, or **Custom Fills** (Edit value menu).

Importing & Exporting Memory Files

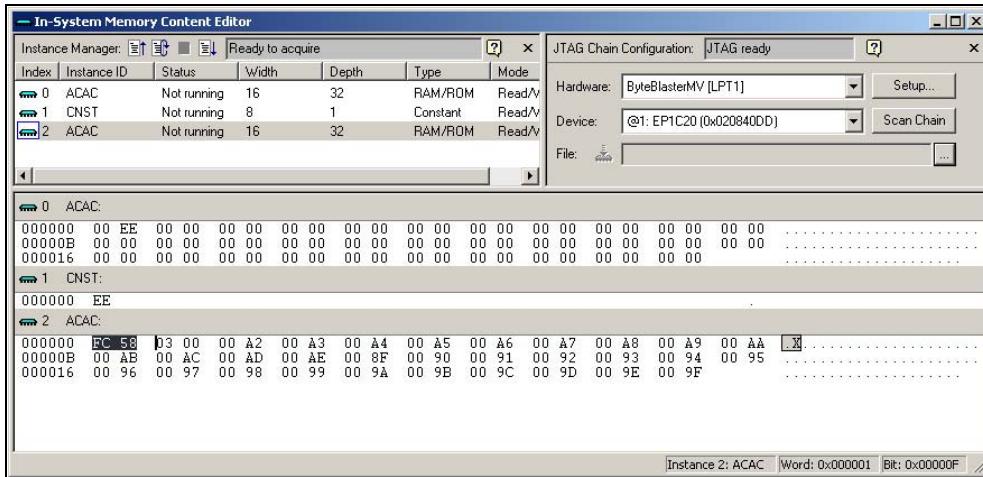
Importing and exporting memory files lets you quickly update in-system memories with new memory images and record data for future use and analysis.

To import a memory file, select an in-system memory or constant from the instance manager and choose **Import Data from File** (Edit menu). You can only import a memory file that is in either a Hexadecimal (Intel-Format) file (.hex) or memory initialization file (.mif) format.

To export data displayed in the Hex Editor to a memory file, select an in-system memory or constant from the instance manager and choose **Export Data to File** (Edit menu). You can export data to a HEX, MIF, Verilog Value Change Dump file (.vcd), or RAM Initialization file (.rif) format.

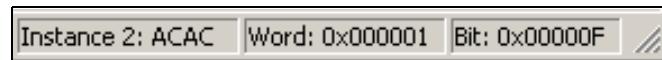
Viewing Memories & Constants in the Hex Editor

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 12–4](#)).

Figure 12–4. 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.

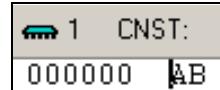
As you analyze the data, you can use the cursor and the status bar to quickly identify the exact location in memory. The status bar is located at the bottom of the In-System Memory Content Editor and displays the selected instance name, word position, and bit offset (Figure 12–5).

Figure 12–5. 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.

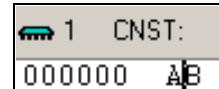
With the cursor in the position shown in Figure 12–6, the word location is 0x0000 and the bit position is 0x0007.

Figure 12–6. Hex Editor Cursor Positioned at Bit 0x0007



With the cursor in the position shown in Figure 12–7, the word location remains 0x0000 but the bit position is 0x0003.

Figure 12–7. Hex Editor Cursor Positioned at Bit 0x0003



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. Choose **In-System Memory Content Editor** (Tools menu).
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, you 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.

Introduction

Beginning with version 4.2, the Altera® Quartus® II software interfaces with EDA tools such as the Cadence Incisive Conformal software and Synplicity Synplify Pro software. Quartus II software supports verifying the functional equivalence between the synthesized (.vqm) netlist from Synplicity Synplify Pro and the post-fit Verilog (.vo) files using Incisive Conformal software. In addition Quartus II software also supports verifying the functional equality between RTL and post-fit VO files from Quartus II using Incisive Conformal.

This chapter discusses the following topics:

- Formal Verification
- Formal Verification Support
- RTL Coding for Quartus Integrated Synthesis (QIS)
- Generating the VO File & Incisive Conformal Script
- Quartus II Scripts for LEC
- Comparing Designs Using Incisive Conformal Software
- Known Issues & Limitations

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 chapter discusses equivalence checking.

The formal verification flow supports Stratix®, Stratix GX, Cyclone™ device families.

Equivalence Checking

Equivalence checking is used to compare the logical equivalence of the original design and the revised design by means of mathematical techniques, rather than by performing simulation using test vectors greatly decreasing the verification cycle of the design.

Formal Verification Support

Altera supports formal verification, using Incisive Conformal, for the following two synthesis tools:

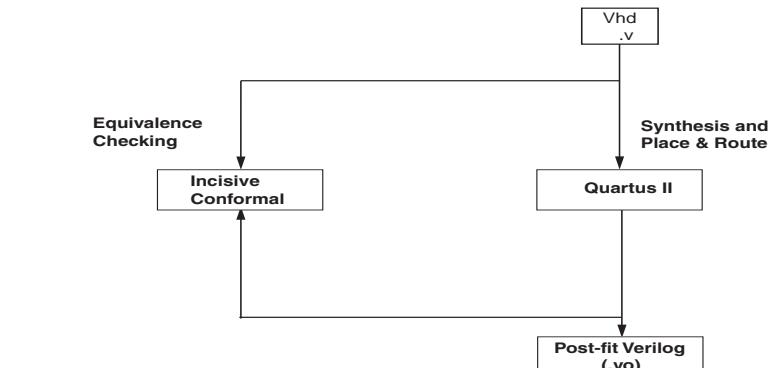
- Quartus Integrated Synthesis
- Synplicity Synplify Pro

The following sections explain how to perform formal verification for each of the synthesis tools.

Quartus Integrated Synthesis

Quartus Integrated Synthesis (QIS) formal verification using RTL and the post-fit VO netlist generated by the Quartus II software is performed as shown in [Figure 13–1](#).

Figure 13–1. Formal Verification Using QIS



EDA Tools Support for QIS

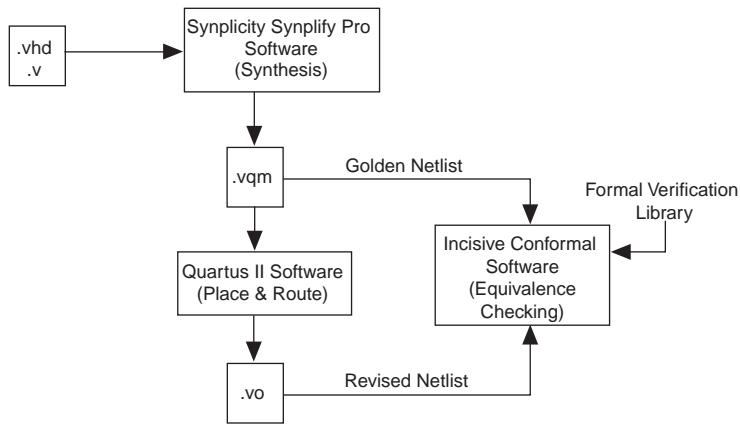
The formal verification flow using the Quartus II software and Cadence Incisive Conformal is supported for the following software versions and platforms:

- Quartus II Software, beginning with version 4.2
- Cadence Software: Incisive Conformal, beginning with 4.3.5.A
- Platform Support: Solaris and Linux

Synplify Pro

When using Synplify Pro, you can use the post-synthesis netlist from Synplify Pro and the post-place-and-route netlist from the Quartus II software for function equivalence with Incisive Conformal software as shown in [Figure 13–2](#).

Figure 13–2. Formal Verification Flow Using Synplify & Incisive Conformal Software



EDA Tools & Device Support for Synplify Pro

The formal verification flow using the Quartus II software, Synplicity Synplify Pro, and Cadence Incisive Conformal is supported for the following software versions and platforms:

- Quartus II Software, beginning with version 4.2
- Cadence Software: Incisive Conformal, beginning with 4.3.5.A
- Synplicity Software: Synplify Pro, beginning with 8.0
- Platform Support: Solaris and Linux

RTL Coding for Quartus Integrated Synthesis (QIS)

Incisive Conformal compares the RTL against post-fit Verilog netlist (VO) generated by the Quartus II software. Incisive Conformal and QIS handle the RTL description in slightly different ways. Features that are supported by the Quartus II software are not always supported by Incisive Conformal and vice versa. The style of the RTL code is of particular concern because some constructs are not supported by both tools, leading to potential formal verification mismatches; one example is state machine extraction. Therefore, for successful verification, both tools need to interpret the RTL code in the same way.

This section provides detail on the problems that might arise and how to prevent them in the formal verification flow. Altera recommends you read, *Recommended HDL Coding Styles* chapter in Volume 1 of the *Quartus II Handbook* for more details on RTL coding styles for Quartus Integrated Synthesis (QIS).

Synthesis Directives & Attributes

Synthesis directives, also known as pragmas play an important role in successful verification of RTL against the post-fit Verilog VO netlist from Quartus II software.

Pragmas that are supported by both; Quartus Integrated Synthesis and Conformal LEC, are supported in the formal verification flow. The same holds for trigger keywords. The trigger keywords “synthesis” and “synopsys” are supported by both tools. Keywords that are not recognized by the Quartus II software such as “verplex” are disabled by the Conformal LEC script (.ctc) file. Use caution with unsupported pragmas because they might lead to verification mismatches.

For example, the RTL code with the synthesis directive `read_comments_as_HDL` can be synthesized using QIS.

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
```

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 above synthesis directive has no affect on Incisive Conformal because it does not support the directive `read_comments_as_HDL`.



For a list of supported attributes and keywords, refer to *Recommended HDL Coding Styles*, chapter, Volume 1 of the *Quartus II Handbook*.

Stuck-at Registers

The QIS eliminates registers that have their output stuck-at a constant value. QIS gives a warning message and adds an entry to the corresponding report panel in the Formal Verification sub-folder of the Analysis & Synthesis section of the Compilation Report. If Conformal LEC does not find the same optimizations, it could lead to unmapped points in the golden netlist. The following example illustrates the issue.

```
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 & g;// e is stuck at 0
    g <= c & e;// g is stuck at 0
    f <= e | b;
  end
  assign out = f & d;
endmodule
```

In the module description above, registers e and g are tied to logic 0. In such a situation, 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
```

Then, QIS adds a command to the formal verification script (.ctc) file telling Conformal LEC that a certain register is stuck at a constant value as shown below.

```
// 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; indicating to Incisive Conformal that a register is stuck at a constant value, which could possibly hide a compilation error. You must verify the design and remove the comments to obtain accurate results.



Altera recommends recoding your design to eliminate “stuck at” registers.

ROM & Shift Register Inference

Because Incisive Conformal cannot infer a LPM megafunction for ROM and shift register, QIS implements both ROM and shift registers in the form of LEs. Using LEs might 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, that indicates that the megafunction wasn't inferred. QIS 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 choose to black box. If the corresponding megafunction is black-boxed before synthesis, the megafunction can be verified by Incisive Conformal.

Latch Inference

A latch is implemented in QIS using a combinational feedback loop instead of using a DLAT (latch primitive in the incisive conformal library) by Incisive Conformal. This results in verification mismatches. The Quartus II software issues a warning message whenever a latch is inferred. The Quartus II software adds a report panel to the Formal Verification sub-folder of the Analysis & Synthesis report. If you cannot eliminate the latch from the design, we recommend that you avoid latches in your design and if it is inevitable we suggest the use of the corresponding megafunction `LPM_LATCH`.



For more information on the problems related to latches, refer to *Recommended HDL Coding Styles* chapter in Volume 1 of the *Quartus II Handbook*.

Combinational Loops

If the design consist of an intended combinational loop, then you must define an appropriate cut point for both the RTL and the post-fit Verilog VO netlist. A warning about the combinational loop that exists in the design is found in the formal verification sub-folder of the Quartus II software analysis and synthesis report.

Finite State Machine Coding Styles

When a state machine is inferred by Incisive Conformal, it uses sequential encoding as the default encoding when no user encoding is present. The Quartus II software selects the encoding that is most suited for the inferred state machine if the **State Machine Processing Settings** on the Analysis & Synthesis settings page of the **Settings** dialog box. is set to the default value which is **Auto**. Therefore, it is important to use the coding style described in the *Recommended HDL Coding Styles* chapter in Volume

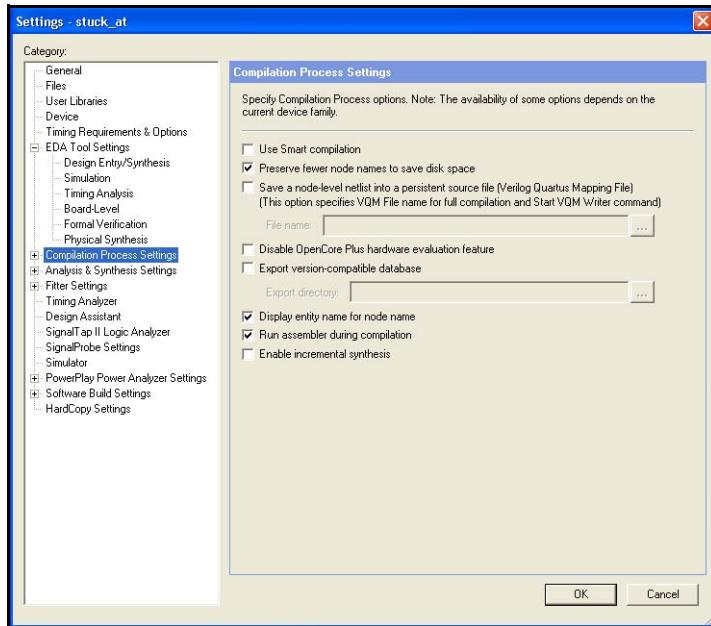
1 of the *Quartus II Handbook* on RTL when writing Finite State Machines (FSMs). In this way, you are guaranteed that both tools (QIS and Incisive Conformal) infer a state machine for the same RTL code. You could choose another encoding such as Minimal Bits, One-Hot, or User-Encoded. In those cases, Quartus II writes the encoding relation between the Conformal LEC encoding and the Quartus II encoding to a file in the `<proj_dir>/db` directory. There is one file for each state machine. The names of the files are `<proj_name>.<number>.fsm`. Quartus II also adds an entry to the formal verification script (`.ctc`) file instructing Conformal LEC to read the file and take the encoding relation into account when verifying the state machine logic.

Generating the VO File & Incisive Conformal Script

The following steps describe how to set up the Quartus II software environment to generate the post-fit VO netlist and Incisive Conformal script for use in formal verification. The steps are identical to both of the Synthesis tools (except for step 3, see the details below):

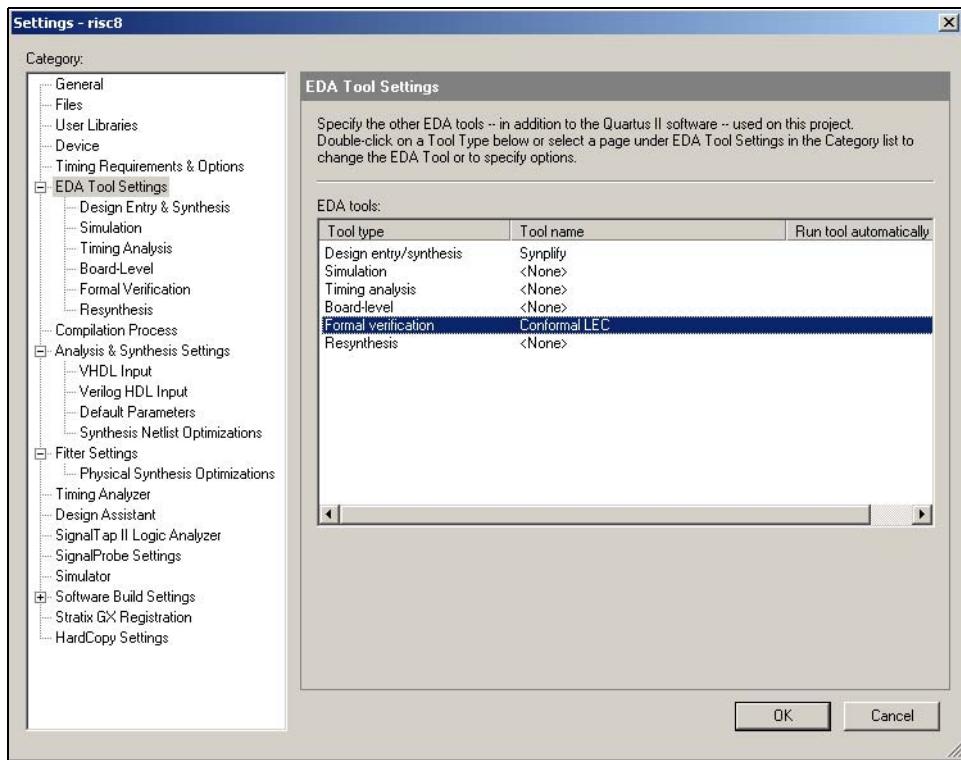
1. If you have not yet done so, create a new Quartus II project or open an existing project.
2. Choose **EDA Tools Settings** (Assignments menu).
3. For QIS, perform the following instructions for Step 3:

On the **EDA Tool Settings** page in the **Settings** dialog box, under **EDA tools**, for **Design Entry/Synthesis** specify **None**. Specify **Conformal LEC** for Formal verification. ([Figure 13–4](#)). On the **Compilation Process settings** page, make sure **Enable incremental synthesis** is turned off as shown in the [Figure 13–3](#).

Figure 13–3. Compilation Process Settings

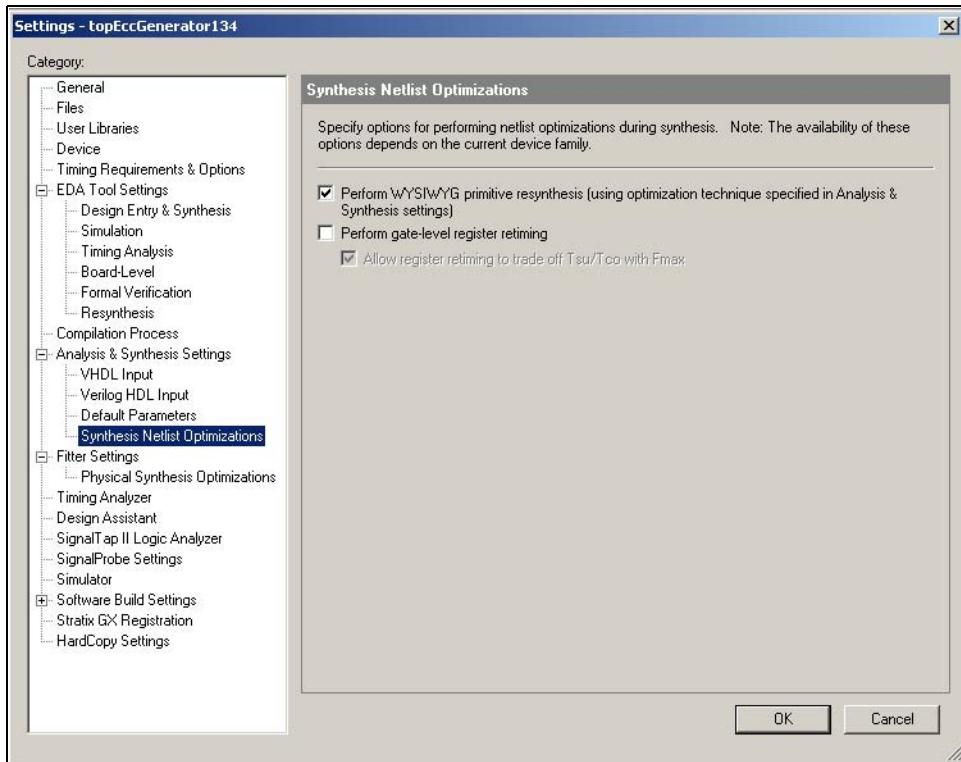
For Synplify Pro, perform the following instructions for Step 3.

On the **EDA Tool Settings** page of the **Settings** dialog box, under **EDA tools**, for **Design Entry/Synthesis** specify **Synplify Pro**. Specify **Conformal LEC** for Formal verification ([Figure 13–4](#)).

Figure 13–4. EDA Tools Selection Note (1)**Note to Figure 13–4:**

- (1) The Quartus II software allows up to six EDA tools to be selected in the EDA tools list.

4. Choose **Analysis & Synthesis** in the **Category** list of the **Settings** dialog box.
5. Under **Analysis & Synthesis**, select **Synthesis Netlist Optimizations**. On the **Synthesis Netlist Optimizations** page, ensure that **Perform gate-level register retiming** is turned off. See **Figure 13–5**.

Figure 13–5. Synthesis Netlist Optimizations

6. Choose **Fitter Settings** in the **Category** list of the **Settings** dialog box. Under **Fitter Settings**, select **Physical Synthesis Optimizations**. On the **Physical Synthesis optimizations** page, ensure that **Perform register retiming** is turned off. See [Figure 13–6](#).

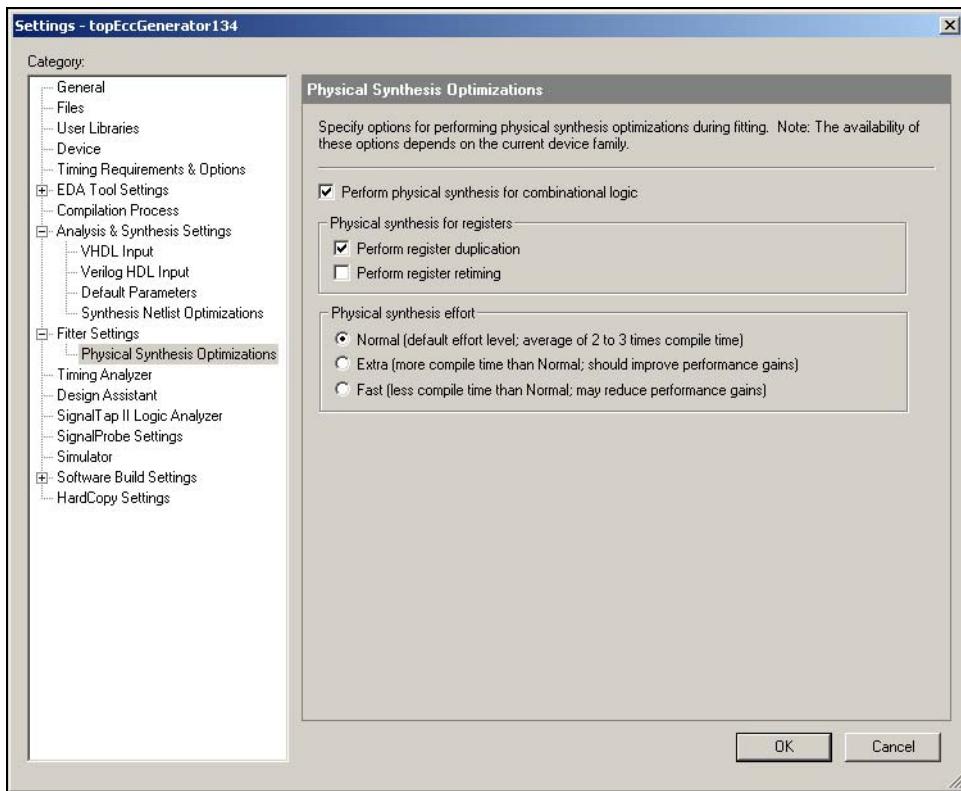


Retiming a design usually results in moving and merging registers along the critical path and is not very well supported by equivalence checking tools. Because equivalence checkers compare the cones of logic terminating at registers, it is necessary that the registers not be moved during Quartus II optimization.

If the options such as **Perform register retiming** (see [Figure 13–5](#)) and **Perform gate-level register retiming** (see [Figure 13–5](#)) are not turned off, the Incisive Conformal script could show up with a different set of compare points, and the resulting netlist is difficult to compare against the reference netlist file.

Beginning with version 4.2, the Quartus II software supports register duplication to improve performance. The Quartus II software generated scripts contain information about the duplicated registers enabling successful formal verification flow. See [Figure 13–6](#).

Figure 13–6. Setting Parameters for Netlist Optimizations



To learn more about register duplication, see the “Physical Synthesis for Registers - Register Duplication” section in the *Netlist Optimization & Physical Synthesis* chapter in Volume 2 of the *Quartus II Handbook*.

7. Perform a full compilation of the design either by selecting **Start Compilation** (Processing menu) or by clicking the **Start Compilation** icon in the tool bar.

If your golden netlist (VQM netlist from Synplify Pro or RTL for QIS) includes any of the following design entities that doesn't have a corresponding formal verification model, then the entity is handled as a black box whose boundary interface is preserved:

- Altera library of parameterized modules (LPMs) functions. The black-box property is applied to only those LPM modules for which an equivalent Incisive Conformal model does not exist.
- Encrypted intellectual property (IP) cores.
- Entities that are defined in the design format other than Verilog HDL or VHDL.

Beginning with version 4.0, the Quartus II software can identify black-boxes automatically and set the **Preserve Hierarchical Boundary** logic option to **Firm** to preserve the boundary interfaces of the black boxes to aid in the formal verification.

You can also set the black-box property on the entities that do not need to be compared by the formal verification tool. To do this make the following assignments either using Tcl commands or GUI for the entities in question:

Tcl Command

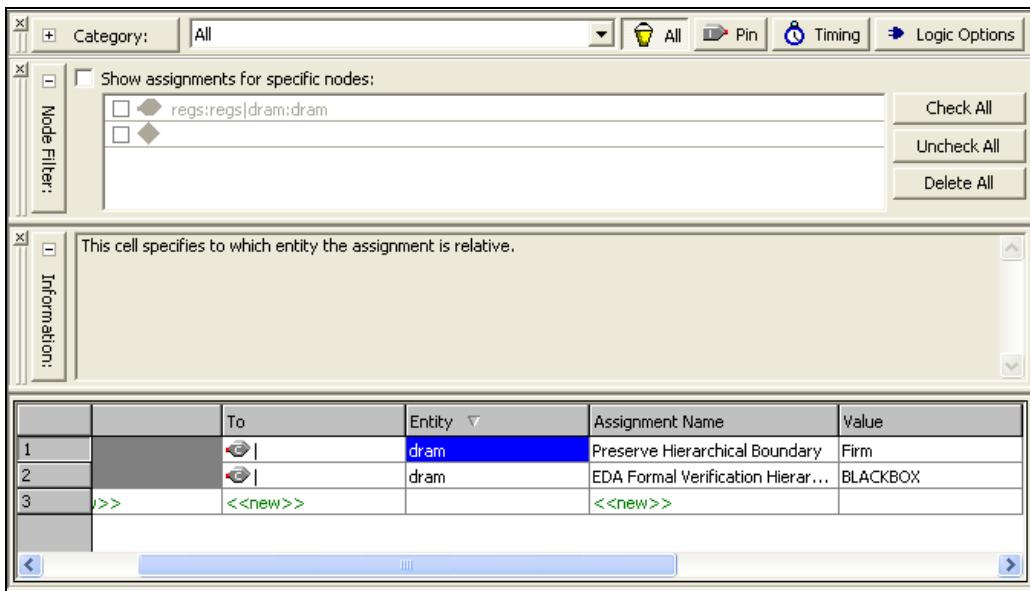
The following two Tcl commands preserves the boundary interface of a black box entity: dram.

```
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 the GUI.

- Make an **EDA Formal Verification Hierarchy** assignment to the entity with the value **BLACKBOX**
- Make a **Preserve Hierarchical Boundary** assignment to the entity with the value **Firm** ([Figure 13-7](#))

Figure 13–7. Setting the Black-Box Property on a Module

The Quartus II software compiler generates:

- A VO file with the name `<design_name>.vo`
- A Script file with the name `<design_name>.ctc` used with Incisive Conformal software, that references `<design_name>.clg` and `<design_name>.clr` which read the library files and black-box descriptions
- A **blackboxes** directory, that contains all the user-defined black box entities in the design in the `<project directory>/fv/conformal/blackboxes` directory

The script file contains the setup constraints to be used along with the formal verification tool. The file `<entity>.v` in the **blackboxes** directory contains the module description of only those entities that are not defined in the formal verification library. 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.

Quartus II Scripts for LEC

The Quartus II software generates scripts to use with Incisive Conformal. This section elaborates the details of the Incisive Conformal commands used within the scripts to help customers in comparing the revised netlist with the golden netlist. Customers in most of the cases need not have to add any more Incisive Conformal constraints to verify the netlists.

Incisive Conformal Commands within the Quartus II Software Generated Scripts

```
setenv QUARTUS <Quartus Installation Dir>
```

This value for the variable QUARTUS is the path to the Quartus II software installation directory.

```
setenv PROJECT <Quartus Project Dir>
```

The Quartus II software assigns the current working directory of the project to the PROJECT variable. Using this variable gives you the flexibility of changing the project directory to the directory where the design files are installed when moving from a UNIX to a Windows environment or the other way around.

```
read design <design files>
```

This command reads both the Golden and the revised netlists along with the appropriate library models:

```
add renaming rule <rule>
```

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 this command the Quartus II software defines temporary substitute string patterns, thus enabling Conformal to automatically map key points when names are not the same.

```
set mapping method <mapping_rule>
```

Conformal LEC software employs three name-based methods to map key points to compare the revised netlist with the golden netlist. Scripts set the right method to get the best results.

```
set flatten model <flattening_rule>
```

The Quartus II software performs several optimizations, one of which is 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`.

```
add black box <module_name>
```

Using the above command, make sure that the following modules are black boxed in both the Golden and revised netlists.

- LPM functions
- Encrypted IP functions
- Entities not implemented in Verilog HDL or VHDL

```
set multiplier implementation <implementation_name>
```

Using the above command you can set the same implementation on multipliers for both the golden and revised netlist.

```
add cut point
```

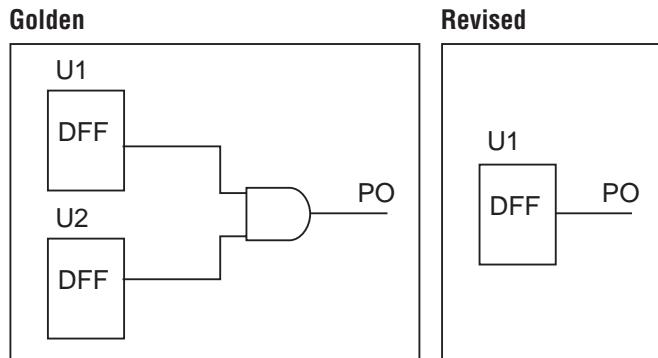
If there are any combinational loops or instances of LPM_LATCH, then the Quartus II software cut the loop at the same point using the above command on both the golden and revised netlists.

```
add mapped points <key_points>
```

Sometimes Incisive Conformal software may not be able to automatically map all the keypoints or may incorrectly map some keypoints. To help Incisive Conformal complete the mapping process, the Quartus II software records optimizations done to the netlist as a series of add mapped points in the Incisive Conformal script.

```
add instance equivalence <instance_name>
```

Figure 13–8. Instance Equivalence



SETUP > add instance equivalence U1 U2 -golden

During the process of optimization, Quartus II software might merge two registers into one as shown in [Figure 13–8](#). Using this command, the Quartus II software tells the formal verification tool that the U1 and U2 registers are equivalent to each other.

```
add instance constraint <constraint_value>
```

At times the register output must be 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 on this command, refer to “[“Stuck-at Registers”](#) on page 13–5.

```
add instance equivalences <instance_pathname>
```

There are times when the inverter is moved beyond the register. When the Quartus II software performs inverter pushback, a command is added to the script to make sure that there is an inverted equivalence to the representative instance.

Comparing Designs Using Incisive Conformal Software

This section discusses using the Incisive Conformal software to compare designs.

Black Boxes in the Incisive Conformal Flow

A module must be treated as a black box by the Incisive Conformal software if the corresponding formal verification model is not available. As discussed in “[“Generating the VO File & Incisive Conformal Script”](#) on page 13–7, the netlist synthesized by the Quartus II software contains black boxes if your project includes any of the following entities:

- LPM functions that do not have Formal Verification models
- Encrypted IP functions
- Entities not implemented in Verilog HDL or VHDL

All LPM functions that do not have Formal Verification models are treated as black boxes by the Synplify software or by QIS. If a corresponding Incisive Conformal verification model exists, however, the LPM function is replaced by logic cells in the VO netlist file generated by the Quartus II software. For example, if the design has references to the functions `lpm_mult` and `lpm_rom`, only `lpm_rom` is treated as a black box, because the corresponding Incisive Conformal verification model is not available.

VO 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 13–5](#)).

If the two assignments listed above are not made for a module, the Quartus II software replaces the black box with logic cells and the VO netlist file no longer contains the black box hierarchy or preserves the port interface, resulting in a mismatch within the Incisive Conformal software.

Running the Incisive Conformal Software

Run Incisive Conformal software from either a system command prompt or using the graphical user interface (GUI), using the CTC script generated by the Quartus II software.

Running the Incisive Conformal Software From a System Command Prompt

To run the Incisive Conformal Software type the following command at a system command prompt:

```
lec -dofile /<path to project directory>/fv/conformal/<design_name>.ctc -nogui ↵
```

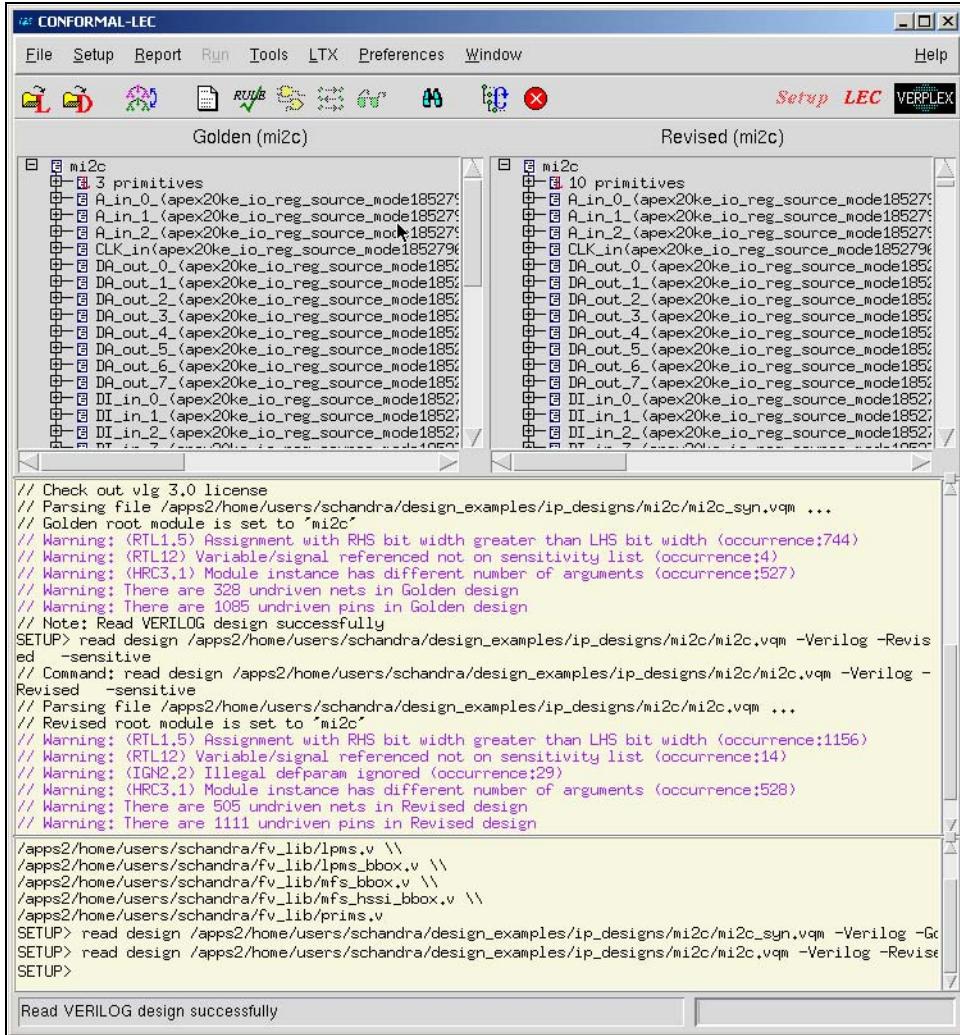
Running the Incisive Conformal Software from the GUI

To run the Incisive Conformal software with the GUI, perform the following steps:

1. Select **Do Dofile** (File menu).
2. Select the file `<path to project directory>/fv/conformal<design>.ctc`.

The Incisive Conformal GUI displays results as shown in [Figure 13–9](#). The original VQM netlist is displayed in the Golden window and the Quartus II-generated VQM netlist is displayed in the Revised window. The status bar at the bottom of the window reports verification results, including the number of compared D-Type Flip-Flops (DFFs) and Primary Outputs (POs), as well as the number of DFFs and POs that are equivalent and non-equivalent, respectively.

Figure 13–9. Incisive Conformal Software GUI Display of Functional Comparisons



To investigate verification results, click the **Mapping Manager** icon in the toolbar, or choose **Mapping Manager** (Tools menu). The Incisive Conformal software reports the mapped, unmapped, and compared points in the Mapped Points, Unmapped Points, and Compared Points windows, respectively.



For more information on how to diagnose non-equivalent points, refer to the user documentation for the Incisive Conformal software.

Known Issues & Limitations

The following known issues and limitations might be encountered when using the formal verification flow described in this chapter:

- 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.
- In designs with combinational feedback loops, the Incisive Conformal software might insert extra cut points in the revised netlist leading to unmapped points and ultimately to verification mismatches. For more information on how to handle combinational loops see “[Combinational Loops](#)” on page 13–6.
- 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 might have an impact on the area and performance.

Conclusion

Formal verification software enables verification of the design during all stages from RTL to placement and routing. Verifying designs takes more times as designs get bigger. Formal verification is a technique that helps reduce the time needed for your design verification cycle.

qii53015-1.0

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 DC FPGA post-synthesis netlist and the 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
- Generating the post-synthesis Verilog file
- Generating the post-place-and-route Verilog (.VO) and Formality script
- Known issues and limitations
- Design comparison using Formality software

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

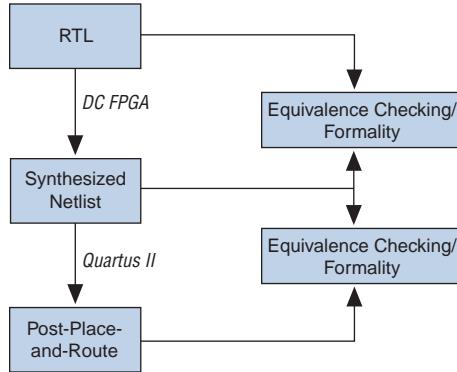
Equivalence checking compares the logical equivalence between the original design and the modified or revised design using mathematical techniques. This method reduces the verification time several-fold compared to the traditional method of performing verification using test vectors. Using a formal verification methodology provides the following key advantages:

- Faster time-to-market
- No test benches or test vectors
- Results in hours compared to days using traditional verification methods

Formal Verification Support

Quartus II software supports formal verification using Formality software for the DC FPGA Synthesis tool as shown in [Figure 14-1](#).

Figure 14-1. Equivalence Checking in the FPGA Design Flow



EDA Tools & Device Support

The formal verification flow using the Quartus II software and Synopsys Formality 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 software and Synopsys Formality supports Solaris and Linux platforms, and supports Stratix series devices.

Formal Verification Between RTL & 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

The changes caused by these synthesis operations cause comparison point matching issues and false verification failures. In order to make sure that the Formality tool 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

Because Formality 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 14–1](#).

<i>Table 14–1. Commands and Affect of Each Command</i>	
Command	Affect
<code>set verilogout_write_constant_nets true</code>	Add this command at the beginning of the script to allow unconnected nets to be driven by either power or ground.
<code>change_names -rule verilog -hierarchy</code>	This command must be added after the compile command to set the Verilog naming rule to the output netlist for all levels of hierarchy.
<code>set_verification_friendly_mode -filename \ <top_level>.svf -append -allow_override</code>	This command helps DC_FPGA to write out a <code>.svf</code> constraint file to be read into Formality.
<code>write -hier -f verilog -o \$outputdir/<top_level>.v</code>	This command writes out a Verilog netlist for Formal Verification.

For a sample DC FPGA script that is ready for compilation, see “[Tcl Sample Script](#)” on page [14–13](#).

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).



Refer to the “DC FPGA Software User Guide” for comparing RTL against post-synthesis netlist using the Formality software.

Generating the VO File & 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. Select **Assignments** > **Settings** > **Files** and highlight the input file. Click **Properties** and select **Verilog Quartus Mapping** (Type menu).
3. Select **Design entry/synthesis** in the **Category** list under **EDA Tool Settings**. Select **Design Compiler FPGA** from the **Tool name** list. See [Figure 14–2](#).

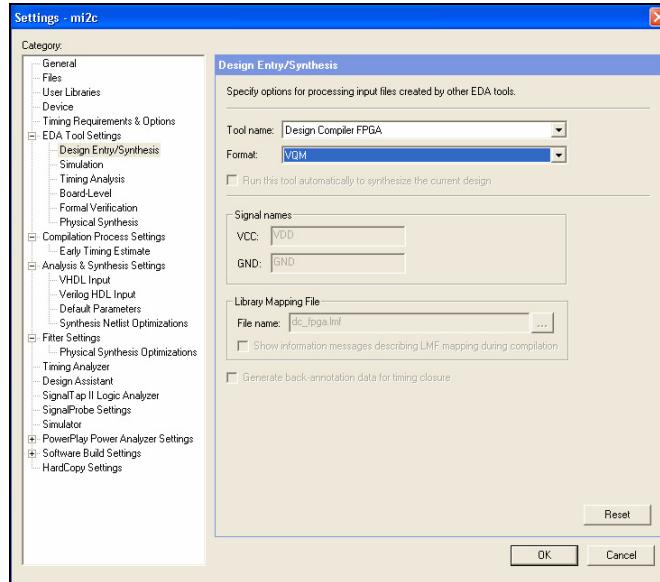
These settings can also be performed using the following Tcl commands:

```
set_global_assignment -name VQM_FILE
<verilog_file_from_dc_fpga>

set_global_assignment -name \
EDA_DESIGN_ENTRY_SYNTHESIS_TOOL "Design Compiler FPGA"

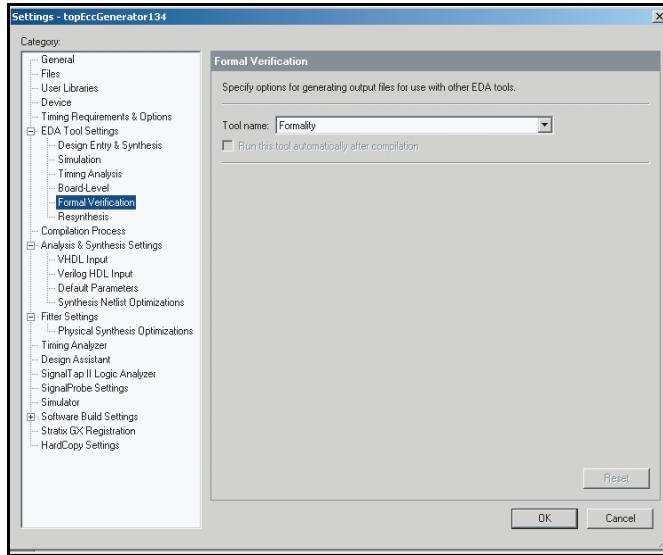
set_global_assignment -name EDA_LMF_FILE \
dc_fpga.lmf -section_id eda_design_synthesis
```

Figure 14–2. EDA Tools Selection

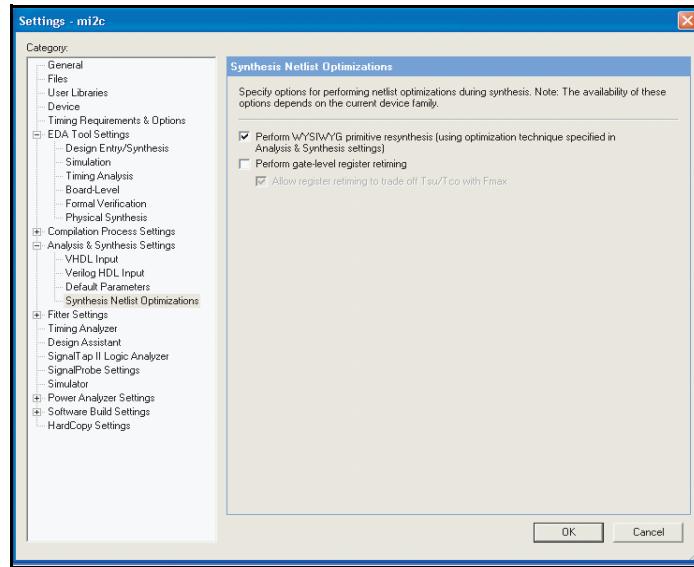


4. Select **Formal verification** in the **Category** list of the Settings dialog box. In the **Tool name** list, select **Formality**. See [Figure 14–3](#).

Figure 14–3. EDA Tools Selection

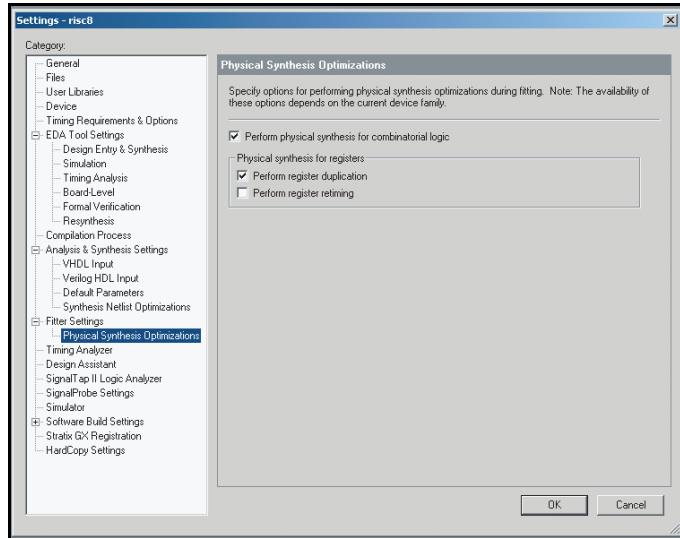


-
5. Click **OK**.
 6. Choose **Settings** (Assignments menu).
 7. In the **Settings** dialog box, expand the Analysis and Synthesis Settings folder and select **Synthesis Netlist Optimizations**.
 8. On the Synthesis Netlist Optimizations page, ensure that the **Perform gate-level register retiming** option is turned off as shown in the [Figure 14–4](#).

Figure 14–4. Synthesis Netlist Optimizations

9. In the **Settings** dialog box, expand the **Fitter Settings** folder and select **Physical Synthesis Optimizations**. On the Physical Synthesis Optimizations page, ensure that the **Perform register retiming** option is turned off. See [Figure 14–5](#).

Figure 14–5. 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, the registers should not moved or duplicated during optimization by the Quartus II software. If the options in this section are not selected, the Formality 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 *Physical Synthesis for Registers-Register Duplication* section in the *Timing Closure Floorplan* chapter in Volume 2 of the *Quartus II Handbook*.

10. Perform a full compilation of the design either by selecting **Start Compilation** (Processing menu) or click 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 that entities that should not be compared by Formality be treated as black boxes. 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.

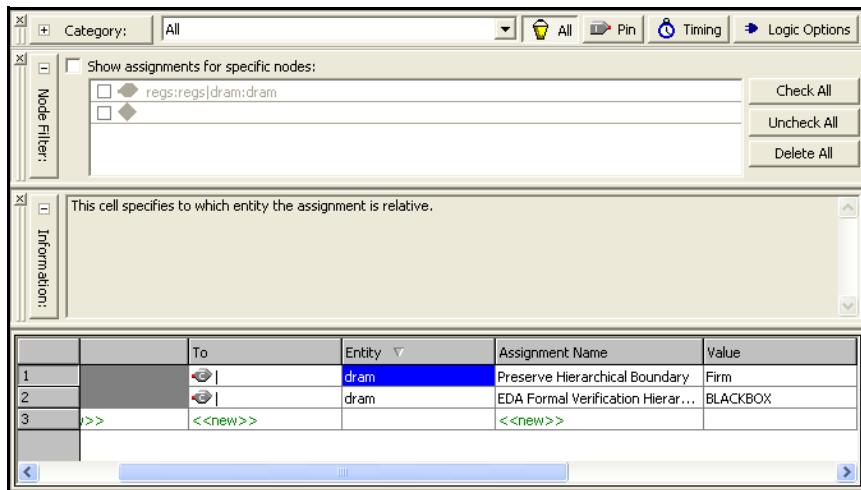
```
set_instance_assignment -name\  
PRESERVE_HIERARCHICAL_BOUNDARY FIRM -to | -entity dram  
set_instance_assignment -name EDA_FV_HIERARCHY\  
BLACKBOX -to | -entity dram
```

GUI

Preserving the boundary interface of an entity using GUI.

- Assign the EDA Formal Verification Hierarchy value as blackbox.
- and
- Assign the Preserve Hierarchical Boundary assignment with a value of Firm to the entity. See [Figure 14–6](#)

Figure 14–6. 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 Formality. 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 Quartus II software, see “[Tcl Sample Script](#)” section at the end of this chapter.

Quartus II Scripts for Formality

The Quartus II software generates scripts to use with Formality. This section describes the Formality commands used within the scripts to help customers comparing the implementation and reference netlists. [Table 14–2](#) describes the Formality commands within Quartus II generated scripts.

Table 14–2. Formality Commands within Quartus II Generated Scripts

Command	Affect
read_verilog <design files>	This command reads both the Golden and the revised netlists in addition to the appropriate library models.
set_compare_rule <rule>	Adds a name matching rule that Formality 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 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 could not match during its matching process.

Comparing Designs Using Formality Software

To verify the functional equivalence between post-synthesis and post-place-and-route netlists, Altera recommends using 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 command line command:

```
%formality -file <file_name>.fms
```



For more information about using Formality software, refer to the *Formality User Guide*.



Formality does not support inferred RAMs in RTL while performing RTL-to-Gates verification. Therefore, the RTL code that infers RAMs must be black boxed while performing synthesis.

Known Issues & Limitations

This section discusses known issues and limitations of the formal verification flow using DC FPGA software, the Quartus II software, and Formality software:

1. Formal verification of post synthesis versus 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 blackbox, 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 web site: *Using the Quartus II Software with DC FPGA*
www.altera.com/support/software/nativelink/quartus2/eda_view_using_eda.htm.

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 Formality. 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
```

A

- Acquisition Clock
 - Assigning 10–5
- Add Signals
 - Command-Line Mode 3–16
 - GUI Mode 3–17
 - to View 3–27
- Adding
 - New Nodes Without Performing a Full Recompilation 10–23
 - Registers for Pipelining 9–12
 - SignalProbe Sources 9–12
 - Signals in Command-Line Mode 3–16
 - Signals in GUI Mode 3–17
- Advanced Timing Analysis 5–14
 - Reports 5–39
 - Reports Using Tcl Scripts 5–39
- ALM
 - Properties 11–15
- Altera
 - Design Flow with ModelSim-Altera Software 1–3
 - Megafunction Simulation Models 1–5
- Altera Megafunction 3–9
- Analyzer
 - Triggering 10–8
- Applications
 - Common 11–28
- Assigning
 - Acquisition Clock 10–5
 - Data Signals 10–7
 - I/O Standards 9–12
 - Signals to the STP File 10–6
- Assignments
 - Multicycle 5–17, 5–18
 - Multicycle Hold 5–18
 - Multicycle Source 5–19
 - Source Multicycle Hold 5–20
- Asynchronous Memory 5–15

B

- Best Case
 - Timing Analysis 5–34
 - 5–33
 - Reporting 5–34
 - Settings 5–33
- Bird's Eye View 11–9
- Black Boxes 14–8
 - Incisive Conformal Flow 13–16

C

- Captured Data
 - Converting to Other File Formats 10–25
 - Saving 10–25
 - Specific RAM Type 10–27
- cds.lib 3–7
 - Command-Line Mode 3–8
 - GUI Mode 3–8
- Chip Editor 11–4
 - Features 11–4
 - Floorplan 11–4
 - Locating a Node 10–34
 - Locating a Node in the 10–34
 - Using in Design Flow 11–2
- Clock
 - Derived 5–14
 - Frequency
 - Maximum 5–3
 - Hold Time 5–2
 - Inverted Assignments 5–10
 - Inverted Clock 5–10
 - Maximum Frequency (fMAX) 5–3
 - Multiple Domains 5–16
 - Not a 5–11
 - Output Clock Frequency
 - Adjusting 11–25
 - Reduce Skew 5–32
- Requirements
 - Specifying Individual 5–7
- Settings 5–8

- Setup Time [5–1](#)
 - Skew [5–5](#)
 - Reduce [5–32](#)
 - to-Output Delay [5–3](#)
 - Combinational Loops [13–6](#)
 - Command-Line Mode
 - Compilation [3–12](#)
 - Commands
 - Tcl [2–12](#)
 - Common Analysis Flows [8–13](#)
 - Common VCS Compiler Options [2–9](#)
 - Comparing Designs Using Formality
 - Software [14–11](#)
 - Comparing Designs Using Incisive Conformal Software [13–16](#)
 - Compilation
 - Command-Line Mode [3–12](#)
 - Faster [10–23](#)
 - GUI Mode [3–13](#)
 - Compile
 - Project Files & Libraries [3–22, 3–25](#)
 - Source Code & Testbenches [3–12](#)
 - Testbench and Design Files into Work Library [1–7](#)
 - Testbench and Design Files into Work Library
 - Using the ModelSim Command Prompt [1–7](#)
 - Testbench and VHO into Work Library [1–23](#)
 - Confidence Metric Details [8–27](#)
 - Converting a HEX File or MIF to a RIF [2–4](#)
 - Correcting a Design Flaw [11–29](#)
 - Create
 - a Timing Netlist [5–38](#)
 - Complex Triggers [10–16](#)
 - In-System Modifiable Memories and Constants [12–3](#)
 - Mnemonics for Bit Patterns [10–25](#)
 - the HDL Representation of the SignalTap II Logic Analyzer [10–10](#)
 - Create a cds.lib File in Command-Line Mode [3–8](#)
 - Create a cds.lib File in GUI Mode [3–8](#)
 - Create an STP File [10–4](#)
 - Create Libraries [3–21](#)
 - Cut Off
 - Feedback
 - I/O Pins [5–29](#)
 - Read During Write Signal Paths [5–30](#)
 - Cut Path
 - Between Unrelated Clock Domains [5–30](#)
 - Timing [5–31](#)
- D**
- Data
 - Capturing to Specific RAM Type [10–27](#)
 - Data Delay
 - Increase [5–32](#)
 - Data Samples
 - View [10–14](#)
 - DC FPGA
 - Software Settings [14–3](#)
 - Synthesis Script [14–13](#)
 - Default Toggle Rate Assignment [8–12](#)
 - Derived
 - Clock [5–14](#)
 - Design
 - Cycle
 - Estimating Power [7–4](#)
 - Elaborate [3–25](#)
 - Example
 - Preserving Timing [10–38](#)
 - Using SignalTap II Logic Analyzers in SOPC Builder Systems [10–41](#)
 - Flaw
 - Correcting [11–29](#)
 - Language Examples [4–10](#)
 - Resources [8–5](#)
 - Simulating with Memory [3–11](#)
 - Design Resources
 - Global Signals [8–5](#)
 - Duty Cycle
 - Adjusting [11–24](#)
- E**
- Early Timing Estimation [5–36](#)
 - EDA
 - Tools & Device Support for Synplify Pro [13–3](#)
 - Tools Support for QIS [13–2](#)
 - EDA Tools & Device Support [14–2](#)
 - Elaborate
 - Command-Line Mode [3–14](#)
 - Design [3–14](#)
 - GUI Mode [3–15](#)
 - Elements

-
- Cyclone I/O 11–20
 - FPGA I/O 11–18
 - Stratix, Stratix GX, & Stratix II I/O 11–18
 - Embedded
 - Logic Analyzer
 - Creating with MegaWizard Plug-In Manager 10–10
 - Multiple Analyzers in One FPGA 10–22
 - Environment
 - Setting Up 3–5, 3–24
 - Setting Variables 3–6
 - Equivalence Checking 14–1
- ## F
- File Conversion
 - HEX 2–4
 - Formal Verification 14–1
 - Between RTL & Post-Synthesis Netlist 14–2
 - Support 14–2
 - FPGA
 - I/O Elements 11–18
 - Memory
 - Preserving 10–15
 - Resources Used by SignalTap II 10–27
- ## G
- Generating Post-Synthesis Netlist for Formal Verification 14–3
 - Generating the VO File & Formality Script 14–4
 - Glitch Filtering
 - Power Analyzer 8–10
 - GUI Mode
 - Create cds.lib file 3–8
- ## H
- Heat Sink & Thermal Compound 8–4
 - Hex Editor
 - Viewing Memory & Constants 12–7
 - High-Level VO or VHO Simulation
 - Models 4–3
 - Hold Time
 - Slack 5–4
 - Hold Time Violations 5–32
 - Fixing 5–32
- ## I
- I/O
 - Timing 11–29
 - I/O Elements
 - Features in the Resource Property Editor 11–22
 - MAX II 11–21
 - I/O Elements
 - Cyclone 11–20
 - MAX II 11–21
 - I/O Pins
 - Number, Type & Loading of 8–5
 - I/O Standards
 - Assigning 9–12
 - I/O Timing 11–29
 - Importing and Exporting Memory Files 12–7
 - Incisive Conformal 13–16
 - Black Boxes in Flow 13–16
 - Running 13–17
 - Running from Command Prompt 13–17
 - Running from GUI 13–17
 - Script & VO File 13–7
 - Software 13–17
 - In-System
 - Configurable Memory and Constants 12–3
 - Memory Content Editor 12–4, 12–9
 - Updating 12–3
- ## L
- Latch Inference 13–6
 - LE
 - Logic Elements, Multiplier Elements & RAM Elements 8–5
 - LE Properties 11–11
 - LE/ALM
 - Supported Changes 11–12
 - Libraries
 - Create 3–6
 - LPM Function 3–9
 - Libraries
 - Quartus II Timing Simulation 3–24
 - Library
 - Setup 3–7
 - Basic 3–7
 - Licensing, Quartus II 1–29
 - Local PC

- Software Setup 10–31
- Logic Element
 - Properties 11–11
- Low-Level VO or VHO Simulation Models 4–2
- LPM
 - Function & Altera Megafunction Libraries 3–9
 - Models
 - Functional RTL Simulation 1–4
 - Simulation 1–4
- LUT
 - Computing the Mask 11–13
 - Equation 11–13
 - Extended Mode 11–17
 - Mask 11–16
- M**
 - Make Multicycle Hold Assignments 5–32
 - Managing Multiple STP Files 10–28
 - Maximum Delay
 - Input 5–8
 - Output 5–10
 - MegaWizard-Generated File
 - Modifying 1–12, 2–4
 - MIF to RIF 2–4
 - Minimum Timing Analysis
 - 5–33
 - Performing 5–34
 - Reporting 5–34
 - Settings 5–33
 - Mnemonics
 - Creating for Bit Patterns 10–25
 - Mode
 - Extended LUT Mode 11–17
 - of Operation 11–13
 - Register Cascade Mode 11–15
 - Shared Arithmetic Mode 11–17
 - Synchronous Mode 11–14
 - ModelSim
 - Compile
 - Simulation Models into Simulation Libraries Using the GUI 1–17
 - Create Simulation Libraries Using GUI 1–16
 - Quartus II Software Output Files 1–19
 - Modes
 - Operation 3–3
- Multicycle
 - Assignment 5–18
- Multicycle Assignment
 - Typical Applications 5–20
- Multicycle Hold
 - Assignment 5–18
- Multicycle Paths 5–17
 - Multi-Frequency Domains 5–25
 - Offsets 5–24
 - Simple 5–20
- N**
 - NativeLink
 - Using with ModelSim 1–27
 - NC-Sim
 - Flow 3–4
 - Generated Simulation Output Files 3–34
 - Simulation Output Files 3–34
 - NC-VHDL Example
 - Simulating the IP Functional Simulation Model in the NC-VHDL Software 4–13
 - Netlist
 - EDA Tools 11–36
 - Gate-Level Timing Simulation for VCS 2–12
 - Gate-Level Timing Simulation the in Quartus II Software 2–7
 - Generating for Other EDA Tools 11–36
 - Post-Synthesis Simulation 2–5
 - Post-Synthesis Simulation Netlist for Modelsim 1–28
 - Post-Synthesis Simulation Netlist for VCS 2–12
 - Node
 - Entity Assignments 8–11
 - Nodes
 - Select Nodes Reserved for Incremental Routing 10–23
 - Set Number Allocated 10–23
 - nopl.v
 - Compiling 2–4
- O**
 - Output
 - Files

-
- Quartus II Simulation 3–22
Maximum Delay and Output Minimum Delay Assignments 5–10
- P**
- Phase Shift
Adjusting 11–24
of PLL
Adjusting to Meet I/O Timing 11–29
Phase Shift of a PLL to Meet I/O Timing 11–29
Pin-to-Pin Delay 5–3
Pipelining
Adding Registers 9–5, 9–12
PLI Routines
Incorporating 3–27
VCS
Software 2–10
PLL
Mode
Normal 11–24
Properties 11–23
Power
Estimating in the Design Cycle 7–4
Factors Affecting Consumption 8–3
Report File 7–6
Supply Planning 8–27
Power Analyzer
Glitch Filtering 8–10
PowerPlay
Early Power Estimator Spreadsheet 7–1
Power Analyzer Compilation Report 8–26
Power Analyzer Flow 8–6
PrimeTime
Environment 6–3
Generated Files 6–3
Format
Specified Constraint Samples 6–4
Quartus II Settings to Generate Files 6–2
Running 6–5
Sample Timing Report 6–6
Timing Reports 6–6
Project Files & Libraries
Compile 3–22
Properties
PLL 11–23
Property Editor 11–11
- Q**
- Quartus II
Early Power Estimator File 7–6
Megafunction
Simulation Models 1–5
Scripts for Formality 14–11
Scripts for LEC 13–14
Settings for Generating PrimeTime Files 6–2
Simulation Output Files 3–20
Software
Licensing & Licensing Set-Up 1–29
Software & NC Simulation Flow
Overview 3–4
Software Generated Formal Verification
Script 14–14
Software Output Files for use in the Model-Sim-Altera Software 1–19
Timing Simulation Libraries 3–24
- R**
- Remote
Debugging Using the SignalTap II Logic Analyzer 10–29
PC
Software Setup 10–30
Requirements
Software 2–1
ROM & Shift Register Inference 13–6
RTL Simulation 2–3
Altera Memory Blocks 2–3
Command-Line Mode 3–19
GUI Mode 3–20
Libraries 1–4
- S**
- Sample Depth
Specifying 10–7
SDF Command File 3–26
Signal Activity File 8–14
SignalProbe
Adding Sources 9–3, 9–12
Compilation 9–7
Performing 9–7
Enable or Disable All 9–13
Fitting Results Modification 9–14
Modify Fitting Results 9–14

- Pins **9–11**
 - Reserving **9–2, 9–11**
- Results Compilation **9–10**
- Routing
 - Enable or Disable All **9–13**
- Routing Failures **9–8**
- Run Automatically **9–13**
- Run Manually **9–13**
- Running with Smart Compilation **9–8, 9–13**
- Smart Compilation **9–8**
 - Using **9–1**
- SignalTap II
 - Analysis
 - Programming Device **10–13**
 - Local PC Setup **10–32**
 - Logic Analyzer
 - Compiling Design **10–9**
 - Creating HDL Representation **10–10**
 - Debug Multiple Designs **10–34**
 - Including in Design **10–3**
 - Instantiating in HDL **10–13**
 - Timing Preservation **10–33**
 - Logic Analyzer in Your Design **10–3**
 - Logic Analyzers
 - SOPC Builder Systems **10–41**
 - Megafunction Ports **10–13**
 - Remote Debugging **10–29**
 - Used FPGA Resources **10–27**
 - Using in Lab Environment **10–28**
- Simulate
 - Design **3–19**
- Simulation
 - Flow **3–2**
 - Gate Level Libraries **1–19**
 - Gate-Level Timing **2–7**
 - Libraries, Gate Level **1–19**
 - Post-Synthesis **3–20**
 - Post-Synthesis Generating Netlist **2–5**
 - Results **8–9**
 - RTL **2–3**
 - Using the ModelSim Command
 - Prompt **1–24**
 - Slack **5–4**
 - Hold Time **5–4**
 - Software
 - Compatibility **1–2**
 - ModelSim-Altera **1–3**
- Quartus II Software Output Files **1–19**
 - Requirements **2–1**
- Source Code & Testbenches
 - Compile **3–12**
- Spread Spectrum
 - Adjusting **11–25**
- Standard Delay Output File
 - Compiling **3–26**
- State Machine
 - Finite Coding Styles **13–6**
- Statically Link **3–32**
- Statically Link the PLI Library With NC-Sim **3–32**
- STP File
 - Assigning Signals **10–6**
 - Creating **10–4**
 - Using to Create Embedded Logic Analyzer **10–4**
- Stratix, Stratix GX, & Stratix II I/O Elements **11–18**
- Synplify Pro **13–3**

T

- Tcl
 - Command **14–9**
 - Commands **2–12**
 - Executing Script-Based Timing Commands **5–6**
 - Sample Script **14–13**
- tco
 - Requirement Assignments **5–13**
- Testbench
 - Compile into Work Library **1–26**
- Timing
 - Analysis
 - Advanced **5–14**
 - Asynchronous Domains **5–32**
 - Basics **5–1**
 - Analysis Reporting **5–14**
 - Analyzer **5–6**
 - Running **11–36**
 - Assignments
 - Setting Global **5–6**
 - Setting Other Individual **5–8**
 - Cut Path **5–31**
 - Simulation

Gate-Level	1–18, 2–7	Command-Line Interface	2–10
Generating Gate-Level Netlist	2–7	Netlist	
Simulation Netlist		Generating Post-Synthesis	
Gate-Level for VCS	2–12	Simulation	2–12
Simulation		Using in Quartus II Design Flow	2–1
Gate-Level	3–3, 3–22	Verilog	
Timing Analyzer Commands	5–38	Code	
transport_int_delays	2–9	Preparing & Linking C Programs	2–10
transport_path_delays	2–8	Functional RTL Simulation with Altera Mem-	
Trigger		ory Blocks	1–11
Creating Complex	10–16	Simulating Designs	1–8
In	10–20	Simulation Designs	1–24
Levels		Verilog Example	
Number of	10–8	Simulating the IP Functional Simulation	
Out	10–20	Model in the ModelSim	
Using as Trigger In of Another		Software	4–10
Analyzer	10–20	Verilog Functional RTL Simulation with Altera	
Position		Memory Blocks	1–11
Specifying	10–9	Verilog HDL	
Type		3–12	
Basic or Advanced	10–8	Code	3–17
Using External	10–19	veriuser.c	
Trigger Levels	10–8	Modified	3–29
Trigger Type		Original	3–29
Basic or Advanced	10–8	VHDL	
V		3–12	
Variable		Simulating Designs	1–5
LM_LICENSE_FILE	1–30	View	
VCS		Data Samples	10–14
Compile Switches		First Level View	11–5
Common	2–9	Second Level View	11–7
Debugging		Third Level View	11–8
VCS		VirSim	
		Using	2–10



Quartus II Handbook, Volume 4 **SOPC Builder**

ALTERA[®]

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
<http://www.altera.com>

Copyright © 2005 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.



Printed on recycled paper



Chapter Revision Dates

vii

About this Handbook.....

ix

How to Contact Altera	ix
Typographic Conventions	1-x

Chapter 1. Introduction to SOPC Builder

Overview	1-1
Architecture of SOPC Builder Systems	1-1
SOPC Builder Components	1-2
SOPC Builder Ready Components	1-3
User-Defined Components	1-4
Avalon Switch Fabric	1-5
Functions of SOPC Builder	1-5
Defining & Generating the System Hardware	1-5
Creating a Memory Map for Software Development	1-6
Creating a Simulation Model & Testbench	1-6
Getting Started	1-6

Chapter 2. Tour of the SOPC Builder User Interface

Starting SOPC Builder	2-1
Starting a New SOPC Builder System	2-1
Working with SOPC Builder Systems	2-2
System Contents Tab	2-3
Adding a Component to the System	2-4
Specifying Connections, Base Address, Clock & IRQ	2-5
Connection Panel	2-6
.....	2-6
Table of Active Components	2-7
Creating User-Defined Components	2-7
System Dependency Tabs	2-8
System Generation Tab	2-9
System Generation Tab Options	2-10
SDK Option	2-11
HDL Option	2-11
Simulation Option	2-12
Starting System Generation	2-13
Other Tools	2-13
Preferences	2-13

Chapter 3. Avalon Switch Fabric

Introduction	3-1
High-Level Description	3-1
Fundamentals of Avalon Switch Fabric Implementation	3-3
Functions of Avalon Switch Fabric	3-3
Address Decoding	3-4
Data-Path Multiplexing	3-5
Wait-State Insertion	3-6
Latency Capabilities	3-7
Native Address Alignment & Dynamic Bus Sizing	3-8
Native Address Alignment	3-9
Dynamic Bus Sizing	3-10
Wider Master	3-10
Narrower Master	3-11
Arbitration for Multi-Master Systems	3-11
Traditional Shared Bus Architectures	3-12
Slave-Side Arbitration	3-13
Arbitrator Details	3-14
Arbitration Rules	3-15
Fairness-Based Shares	3-15
Round-Robin Scheduling	3-16
Minimum Share Value	3-17
Setting Arbitration Parameters in the SOPC Builder GUI	3-17
Clock Domain Crossing	3-18
Description of Clock Domain-Crossing Logic	3-18
Location of Clock Domain Crossing Logic	3-20
Duration of Transfers Crossing Clock Domains	3-20
Implementing Multiple Clock Domains in the SOPC Builder GUI	3-21
Interrupt Controller	3-21
Software Priority	3-22
Hardware Priority	3-23
Assigning IRQs in the SOPC Builder GUI	3-23
Reset Distribution	3-24

Chapter 4. SOPC Builder Components

Introduction	4-1
Sources of Components	4-1
Location of the Component Hardware	4-2
Components That Include Logic Inside the System Module	4-2
Components That Interface to Logic Outside the System Module	4-3
Structure & Contents of a Component Directory	4-3
class.ptf File	4-3
cb_generator.pl File	4-4
hdl Directory	4-5
Other Component Files	4-5
Component Directory Location	4-6

Chapter 5. Component Editor

Introduction	5-1
Component Editor Output	5-2
Starting the Component Editor	5-3
HDL Files Tab	5-3
Signals Tab	5-4
Interfaces Tab	5-5
SW Files Tab	5-6
Component Wizard Tab	5-8
Identifying Information	5-8
Parameters	5-9
Saving a Component	5-10
Re-Editing a Component	5-11

Chapter 6. Developing Components for SOPC Builder

Introduction	6-1
SOPC Builder Components and the Component Editor	6-1
Assumptions About the Reader	6-2
Hardware and Software Requirements	6-2
Component Development Flow	6-3
Typical Design Steps	6-3
Hardware Design	6-4
Software Design	6-5
Verifying the Component	6-7
Unit Verification	6-7
System-Level Verification	6-7
Design Example: Pulse-Width Modulator Slave	6-8
Install the Design Files	6-8
Review the Example Design Specifications	6-9
PWM Design Files	6-10
Functional Specification	6-10
PWM Task Logic	6-11
Register File	6-12
Avalon Interface	6-13
Software API	6-14
Package the Design Files into an SOPC Builder Component	6-15
Open the Quartus II Project & Start the Component Editor	6-15
HDL Files Tab	6-15
Signals Tab	6-17
Interfaces Tab	6-18
Software Files (SW Files) Tab	6-19
Component Wizard Tab	6-20
Save the Component	6-21
Instantiate the Component in Hardware	6-21
Add a PWM Component to the SOPC Builder System	6-22
Modify the Quartus II Design to Use the PWM Output	6-23
Compile the Hardware Design and Download to the Target Board	6-24

Exercise the Hardware Using Nios II Software	6-24
Start the Nios II IDE & Create a New IDE Project	6-25
Compile the Software Project and Run on the Target Board	6-27
Sharing Components	6-28

Chapter 7. Building Systems with Multiple Clock Domains

Introduction	7-1
Example Design Overview	7-1
Hardware and Software Requirements	7-2
Creating the Multi-Clock Hardware System	7-3
Copy the Hardware Design Files to a New Directory	7-3
Modify the Design in SOPC Builder	7-4
Open the System in SOPC Builder	7-4
Add DMA Controller and Memory Components	7-5
Connect DMA Master Ports to Memory Slave Ports	7-6
Make Clock Domain Assignments	7-7
Update the Quartus II Design	7-8
Update the System Module Symbol	7-8
Update PLL Settings to Generate a 100 MHz Clock	7-10
Connect the 100 MHz clock to the system module	7-12
Compile the Design and Download to the Board	7-14
Running Software to Exercise the Multi-Clock Hardware	7-15
Install the Example Software Design Files	7-16
Create a New Nios II IDE Project	7-16
Build and Run the Program	7-18
Conclusion	7-19



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: February 2005

Part number: QII54001-1.0

Chapter 2. Tour of the SOPC Builder User Interface

Revised: February 2005

Part number: QII54002-1.0

Chapter 3. Avalon Switch Fabric

Revised: February 2005

Part number: QII54003-1.0

Chapter 4. SOPC Builder Components

Revised: February 2005

Part number: QII54004-1.0

Chapter 5. Component Editor

Revised: February 2005

Part number: QII54005-1.0

Chapter 6. Developing Components for SOPC Builder

Revised: February 2005

Part number: QII54007-1.0

Chapter 7. Building Systems with Multiple Clock Domains

Revised: February 2005

Part number: QII54008-1.0



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, go to the Altera world-wide web site at www.altera.com. For technical support on this product, go to www.altera.com/mysupport. For additional information about Altera products, consult the sources shown below.

Information Type	USA & Canada	All Other Locations
Technical support	www.altera.com/mysupport/ (800) 800-EPLD (3753) (7:00 a.m. to 5:00 p.m. Pacific Time)	www.altera.com/mysupport/ +1 408-544-8767 7:00 a.m. to 5:00 p.m. (GMT -8:00) Pacific Time
Product literature	www.altera.com	www.altera.com
Altera literature services	literature@altera.com	literature@altera.com
Non-technical customer service	(800) 767-3753	+1 408-544-7000 7:00 a.m. to 5:00 p.m. (GMT -8:00) Pacific Time
FTP site	ftp.altera.com	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: f_{MAX} , \qdesigns directory, d: drive, chiptrip.gdf file.
<i>Italic Type with Initial Capital Letters</i>	Document titles are shown in italic type with initial capital letters. Example: <i>AN 75: High-Speed Board Design</i> .
<i>Italic type</i>	Internal timing parameters and variables are shown in italic type. Examples: <i>t_{PIA}</i> , <i>n + 1</i> . 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: <code>data1</code> , <code>tdi</code> , <code>input</code> . Active-low signals are denoted by suffix <code>n</code> , e.g., <code>resetn</code> . Anything that must be typed exactly as it appears is shown in Courier type. For example: <code>c:\qdesigns\tutorial\chiptrip.gdf</code> . Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword <code>SUBDESIGN</code>), as well as logic function names (e.g., <code>TRI</code>) 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.
	The caution indicates required information that needs special consideration and understanding and should be read prior to starting or continuing with the procedure or process.
	The warning indicates information that should be read prior to starting or continuing the procedure or processes
↔	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. Chapters in this section serve to answer the following questions:

- What is SOPC Builder?
- What services does SOPC Builder provide?

This section includes the following chapters:

- Chapter 1, Introduction to SOPC Builder
- Chapter 2, Tour of the SOPC Builder User Interface
- Chapter 3, Avalon Switch Fabric
- Chapter 4, SOPC Builder Components
- Chapter 5, Component Editor

Revision History

The following table shows the revision history for Chapters 1–5. These version numbers track the document revisions; they have no relationship to the version of SOPC Builder or the Quartus® II software.

Chapter(s)	Date / Version	Changes Made
1	February 2005 v1.0	Initial release.
2	February 2005 v1.0	Initial release.
3	February 2005 v1.0	Initial release.
4	February 2005 v1.0	Initial release.
5	February 2005 v1.0	Initial release.

QI154001-1.0

Overview

SOPC Builder is a powerful system development tool for creating systems based on 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 and is available to all Altera customers.

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 arbitrary SOPC designs 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 had to manually write top-level HDL files that wire together the pieces of the system. Using SOPC Builder, you specify the system components in a graphical user interface (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.

In addition to its role as a hardware generation tool, SOPC Builder also serves as the starting point for system simulation and embedded software creation. 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.

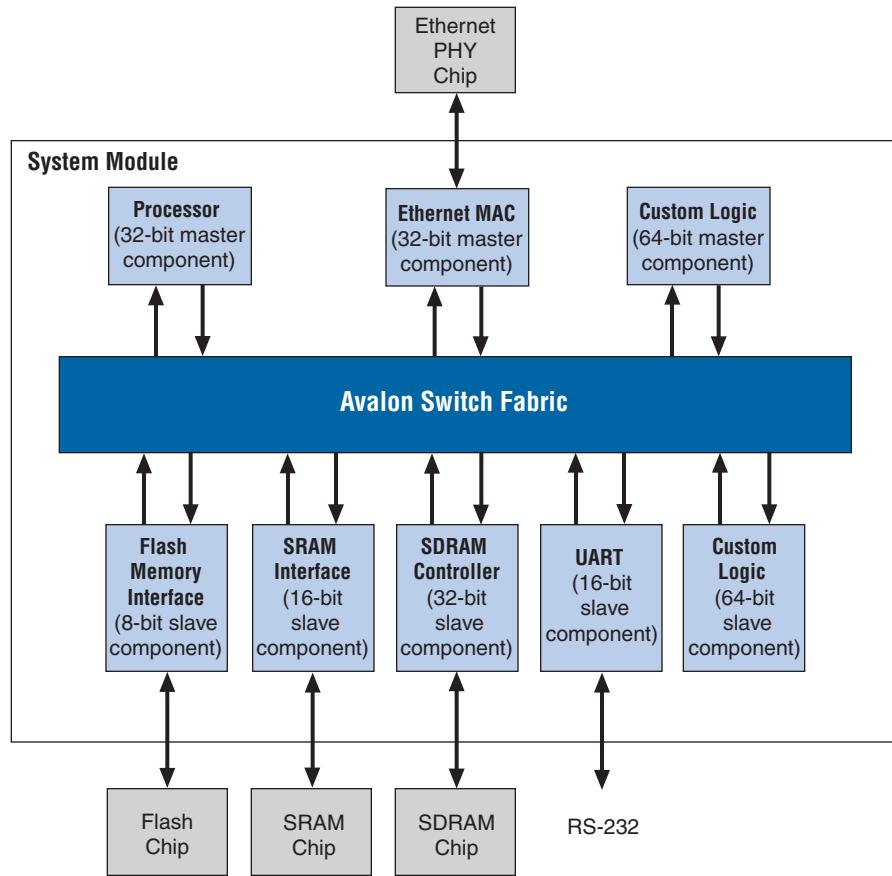
Architecture of SOPC Builder Systems

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. SOPC Builder connects multiple components together to create a top-level HDL file called the *system module*. SOPC Builder generates *Avalon™ switch fabric* that contains logic to manage the connectivity of all components in the

system. [Figure 1–1](#) shows an example of a multi-master system module with Avalon switch fabric connecting the multiple master and slave components.

Figure 1–1. Example of a System Module Generated by SOPC Builder



SOPC Builder Components

SOPC Builder components are the building blocks of the system module. SOPC Builder components use the Avalon interface 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. The Avalon interface uses an address-mapped read/write protocol that allows master components to read and/or write any slave component.



For details on the Avalon interface, see the *Avalon Interface Specification* at www.altera.com.

A component can be a logical device that is entirely contained within the system module, such as the processor component in [Figure 1–1 on page 1–2](#). Alternately, a component can act as an interface to an off-chip device, such as the SRAM interface component in [Figure 1–1 on page 1–2](#). 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 Ethernet MAC in [Figure 1–1 on page 1–2](#).

A component can be instantiated more than once per design.

Altera and third-party developers provide ready-to-use SOPC Builder components, such as:

- Microprocessors, such as the Nios II processor
- Microcontroller peripherals
- 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
- Interfaces to off-chip devices
 - Memory controllers
 - Buses and bridges
 - Application-specific standard products (ASSP)
 - Application-specific integrated circuits (ASIC)
 - Processors

SOPC Builder Ready Components

Altera awards the SOPC Builder Ready certification to intellectual property (IP) designs that have plug-and-play integration with SOPC Builder. These functions may be accompanied by software drivers, low-level routines, or other software design files.

Altera's OpenCore® and OpenCore Plus evaluation programs allow you to "test drive" an SOPC Builder component both in simulation and in hardware before you buy. You can download evaluations of Altera IP functions directly from www.altera.com/IPMegastore. For IP functions provided by third-party vendors, contact the vendor directly to obtain an OpenCore evaluation.



Check the Altera web site at www.altera.com for up-to-date information about available SOPC Builder Ready components. You can identify SOPC Builder Ready components by the logo shown in [Figure 1–2](#).

Figure 1–2. The SOPC Builder Ready Certification Logo



User-Defined Components

SOPC Builder provides an easy method for you to develop and connect your own components. With the Avalon interface, user-defined logic need only adhere to a simple interface based on address, data, read-enable, and write-enable signals.

You use the following design flow to integrate custom logic into an SOPC Builder system:

1. Define the interface to the user-defined component.
2. If the component logic resides on-chip, 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 Ready 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, see the *Developing SOPC Builder Components* chapter in Volume 4 of the *Quartus II Handbook*. For complete detail on the file structure of a component, see the *SOPC Builder Components* chapter in Volume 4 of the *Quartus II Handbook*. For details on the SOPC Builder component editor, see the *Component Editor* chapter in Volume 4 of the *Quartus II Handbook*.

Avalon Switch Fabric

The Avalon switch fabric is the glue that binds SOPC Builder-generated systems together. The Avalon switch 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. SOPC Builder generates the Avalon switch fabric automatically, so that you do not have to manually perform the tedious, error-prone task of connecting hardware modules.

The purpose of SOPC Builder is to abstract away the complexity of interconnect logic, allowing designers to focus on the details of their custom components and the high-level system architecture.

Automatically generating the Avalon switch fabric is the keystone to achieving this purpose. Avalon switch fabric in the system module is like air for humans: Its existence is essential, but largely ignored. Because SOPC Builder generates Avalon switch fabric automatically, most users do not interact directly with it or the HDL that describes it.



For further details, see the *Avalon Switch Fabric* chapter in Volume 4 of the *Quartus II Handbook*.

Functions of SOPC Builder

This section describes the fundamental functions of SOPC Builder.

Defining & Generating the System Hardware

The purpose of the SOPC Builder GUI is to allow you to easily define the structure of a hardware system, and then generate the system. The GUI is designed for the tasks of adding components to a system, configuring the components, and specifying how they connect together.

After you add all components and specify all necessary system parameters, SOPC Builder is ready to generate the Avalon switch fabric and output the HDL files that describe the system. 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)
- (Optional) Software files for embedded software development, such as a memory-map header file and component drivers
- (Optional) Testbench for the system module and ModelSim® simulation project files

After you generate the system module, it can be compiled directly by the Quartus II software, or instantiated in a larger FPGA design.



For more detail on the SOPC Builder GUI for defining and generating systems, see the *Tour of the SOPC Builder User Interface* chapter in Volume 4 of the *Quartus II Handbook*.

Creating a Memory Map for Software Development

For each microprocessor in the system, SOPC Builder optionally generates a header file that defines the address of each slave component. In addition, each slave component can provide software drivers and other software functions and libraries for the processor.

The process for writing 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 & Testbench

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 Development Kit, Evaluation Edition. You can download this kit for free from the Altera Download Center at www.altera.com.

QII54002-1.0

This chapter provides reference on how to access the features available in the SOPC Builder graphical user interface (GUI). This chapter will familiarize you with the main features of the SOPC Builder GUI.



Due to evolution and improvement of the software, the figures in this chapter may not match exactly what you see in the software.

Starting SOPC Builder

Each SOPC Builder system is associated with one Quartus® II project. Therefore, to launch SOPC Builder, you must first open a project in the Quartus II software. With a Quartus II project open, you can launch SOPC Builder by choosing **SOPC Builder** (Tools menu) or by clicking the **SOPC Builder** toolbar button. See [Figure 2-1](#).

Figure 2-1. SOPC Builder Toolbar Button



The SOPC Builder toolbar button might not be displayed by default. To enable it, use the **Customize** window (Tools menu).

Starting a New SOPC Builder System

If an SOPC Builder system does not exist in the current Quartus II project directory, SOPC Builder will display the **Create New System** dialog as shown in [Figure 2-2 on page 2-2](#), and prompt you to specify the following:

- A name for the new SOPC Builder system – This serves as the name of the system module that SOPC Builder will generate.
- Target HDL – This setting determines the output language of the system module.

Figure 2–2. Create New System Dialog



Working with SOPC Builder Systems

SOPC Builder operates on exactly one system at a time, and the system must be associated with a Quartus II project. A Quartus II project may have multiple associated SOPC Builder systems, but typically will have only one (if any).



If you integrate multiple SOPC Builder system modules in one Quartus II project, you must make sure all components are named uniquely across all system modules. Otherwise, filename collision will occur.

In SOPC Builder, you can create a new system by choosing **New System** (File menu). You can switch to a different system by choosing **Open System** (File menu).

SOPC Builder saves files in the same directory as the Quartus II project. Each SOPC Builder system is represented by a file named *<system module name>.ptf*, which is a plain-text file describing the structure of the system and other system-specific details. In a purely mechanical sense, the SOPC Builder GUI is a .ptf file editor.



Changes you make in the SOPC Builder GUI are saved immediately to the .ptf file. When you open a system, SOPC Builder creates a back-up file named *<system module name>.ptf.bak*, in case you need to revert changes.

System Contents Tab

SOPC Builder employs a tabbed user interface. Tasks are categorized by function, and related tasks are presented on the same tab.

The **System Contents** tab is displayed when you open SOPC Builder. It is the view of SOPC Builder that you will use most often. You use the **System Contents** tab to do the following:

- Add components to a system
- Configure the components
- Specify connections between components

Figure 2–3 lists the elements of the **System Contents** tab. See Table 2–1 on page 2–4 for details.

Figure 2–3. Elements of the System Contents Tab

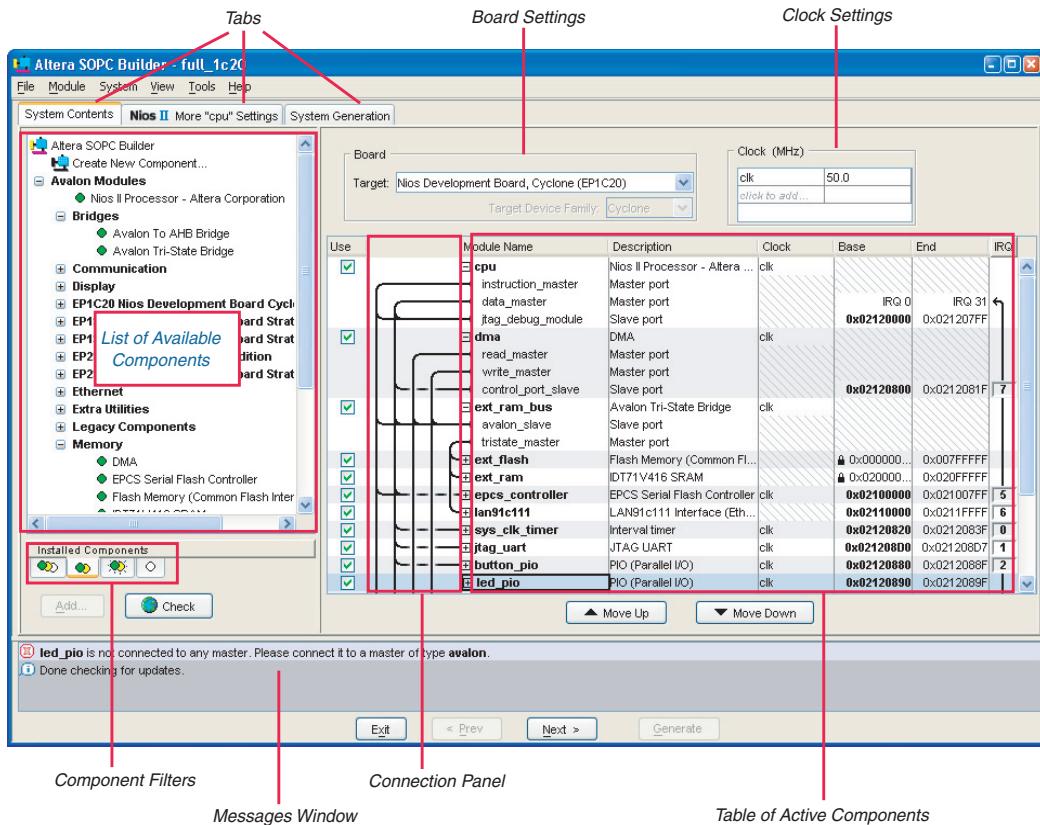


Table 2–1 describes the GUI elements shown in Table 2–1.

Table 2–1. GUI Elements on the System Generation Tab	
GUI Element	Description
Tabs	Categorizes GUI controls, based on task.
List of Available Components	<p>Lists the library of available SOPC Builder components, organized by category. Each component appears with a colored dot next to its name. The colors of the dots have the following meanings:</p> <ul style="list-style-type: none"> ● Green dot – A full, licensed version of the component is installed. ● Yellow dot – A limited, evaluation version of the component is installed. ● White dot – This component is available from Altera or a partner vendor, but is not installed.
Component Filters	Filters which type of components appear in the list of available components.
Table of Active Components (1)	<p>Lists the components instantiated in the current system, and allows you to specify the following:</p> <ul style="list-style-type: none"> ● Name for each component instance ● Base address for each slave port ● Clock source for each component instance ● Interrupt priority (if any) for each slave port
Connection Panel (1)	<p>Represents connections between the components, and allows you to perform the following:</p> <ul style="list-style-type: none"> ● Specify connections between master ports and slave ports. ● Specify arbitration shares for slave ports that are shared by multiple master ports.
Board Settings	Allows you to specify a target board and a FPGA device family. These settings provide SOPC Builder with information about the target hardware, such as pin-outs and connections to off-chip devices.
Clock Settings	Allows you to specify the clock inputs and frequencies to the system module.
Messages Window	Displays information, warning, or error messages related to the current system.

Note for Table 2–1:

(1) Options available in the View menu alter how this element is displayed.

You can connect any master port to any slave port, as long as they use the same interface protocol. If they use different interface protocols, they must communicate through a bridge component, such as the AHB-to-Avalon™ bridge provided with SOPC Builder.

Adding a Component to the System

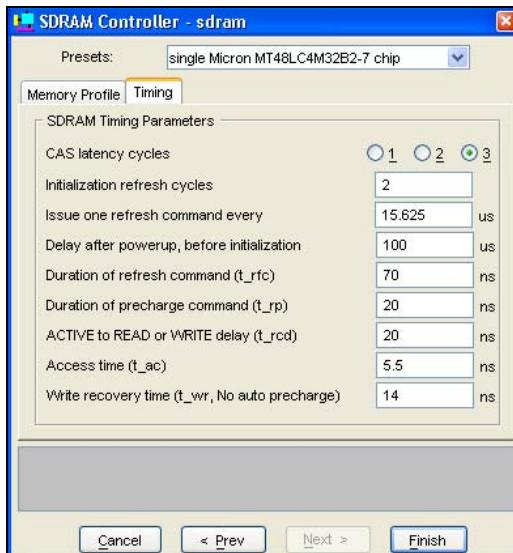
To add a component to the system, find the component in the list of available components and do one of the following:

- Double-click the component.
- Select the component, then click **Add**.

- Right-click the component and choose Add New <component name>.

If the component has any parameterized features, a wizard will appear allowing you to configure this instance of the component. [Figure 2–4](#) shows an example of a configuration wizard for the SDRAM controller that is included in the Nios II development kit.

Figure 2–4. SDRAM Controller Configuration Wizard



Like the overall SOPC Builder GUI, component configuration wizards use tabs to categorize GUI features, and have a message window that displays dynamic information about the current configuration of the component.

The parameters and information displayed in the configuration wizard depend on the component. Many wizards provide tool tips that give you information on how to specify each parameter.



You can open component documentation from within SOPC Builder. Right-click the component in the list of available components, and choose one of the document items listed.

Specifying Connections, Base Address, Clock & IRQ

After you add a component, you must configure how it fits within the system.

Connection Panel

In the connection panel you specify the master-slave connections between components.



Each SOPC Builder component can have one or more master and slave ports. Each slave port must be connected to a master port.

Hovering the mouse over the connection panel displays the potential connection points between components, represented as dots connecting wires. A filled dot shows that a connection is made; an open dot shows a potential connection point that is not currently connected. Clicking a dot toggles the connection status. See [Figure 2–5](#).

Figure 2–5. Connection Panel

Module Name	Description	Clock	Bus Type	Base	End	IRQ
cpu	Nios II Processor - Alter...	clk				
instruction_master	Master port		avalon			
data_master	Master port		avalon			
jtag_debug_module	Slave port		avalon	IRQ 0 0x02120000	IRQ 31 0x021207FF	
sdram	SDRAM Controller	clk	avalon	0x01000000	0x1FFFFFFF	
sys_clk_timer	Interval timer	clk				
s1	Slave port		avalon	0x02120820	0x0212083F	3
jtag_uart	JTAG UART	clk	avalon	0x021208D0	0x021208D7	4
dma/read_master is not a master of sys_clk_timer/s1						

You can connect any master port to any slave port, as long as the ports use the same type of hardware interface. If they use different interfaces, they must communicate through a bridge component, such as the AHB-to-Avalon bridge provided with SOPC Builder.

Table of Active Components

After your component is connected in the connection panel, you use the table of active components to specify the parameters listed in [Table 2–2](#).

Table 2–2. Table of Active Components	
Parameter	Description
Name	Name of the instance of the component. SOPC Builder assigns a default name when you add a new component. To change this name, right click the component name and select Rename .
Use	Enables/Disables the component in the current system. Turning off the Use setting is equivalent to removing the component from the system.
Clock	Clock source for the component. You must choose one of the clocks entered in the clock settings.
Base (1)	<p>Base address where the slave port will appear in the master port's address space. SOPC Builder can automatically choose new base address values to avoid conflicts between all components.</p> <p>You can automatically assign base addresses for all components by choosing Auto-Assign Base Addresses (System menu).</p> <p>You can lock the base address on a component so that SOPC Builder will not change it automatically. To lock the base address, right-click the component in the table of active components, and choose Lock Base Address.</p>
IRQ (1)	<p>Specifies the IRQ value the slave drives to the master, if the slave port can generate interrupts. If you specify NC for an IRQ value, SOPC Builder will not connect interrupt signals from slave to master.</p> <p>SOPC Builder can automatically choose new IRQ values to avoid conflicts between all components. To automatically assign IRQs for all components, choose Auto-Assign IRQs (System menu).</p>
<p><i>Note to Table 2–2:</i></p> <p>(1) This setting applies to slave ports only.</p>	



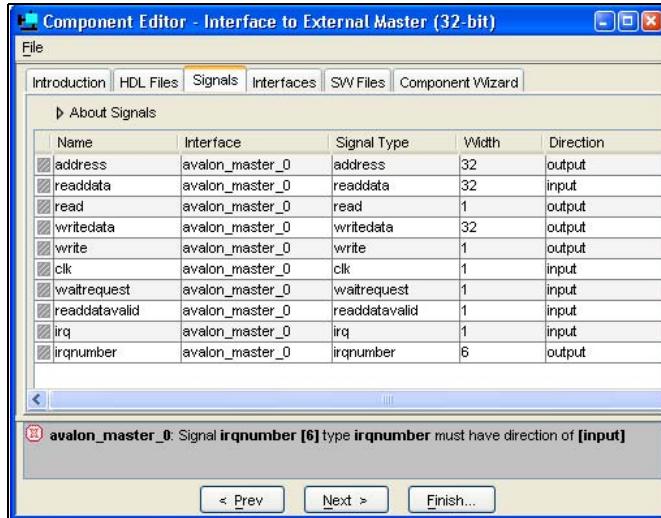
SOPC Builder automatically chooses defaults for these parameters when you add each component, but you must verify that they are appropriate for your system.

Creating User-Defined Components

You can use the SOPC Builder component editor to create a component from user-defined logic. To open the component editor, choose **New Component** (File menu) or select **Create New Component** in the list of available components, and click **Add**.

Figure 2–6 shows an example of the component editor editing a 32-bit master component.

Figure 2–6. SOPC Builder Component Editor



After creating a user-defined component, the process to instantiate the component is the same as for any other component.



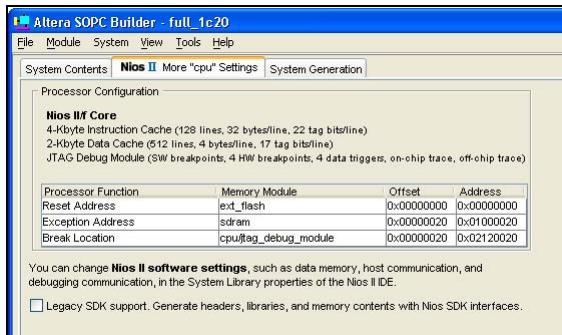
For instructions on developing a custom SOPC Builder component, see the *Developing SOPC Builder Components* chapter in Volume 4 of the *Quartus II Handbook*. For complete detail on the file structure of a component, see the *SOPC Builder Components* chapter in Volume 4 of the *Quartus II Handbook*. For details on the SOPC Builder component editor, see the *Component Editor* chapter in Volume 4 of the *Quartus II Handbook*.

System Dependency Tabs

Certain components must be configured based on parts of the system that are external to the component. SOPC Builder provides system dependency tabs, which allow further configuration of a component beyond the component's configuration wizard. System dependency tabs are titled **More "<Name of component instance>" Settings**, and appear next to the **System Contents** tab.

Figure 2–7 shows an example of the system dependency tab for an instance of the Nios II processor named `cpu`. In this example, the Nios II processor reset and exception addresses depend on memory components in the system but external to the processor, and therefore uses a system dependency tab.

Figure 2–7. System Dependency Tab for a Nios II Processor

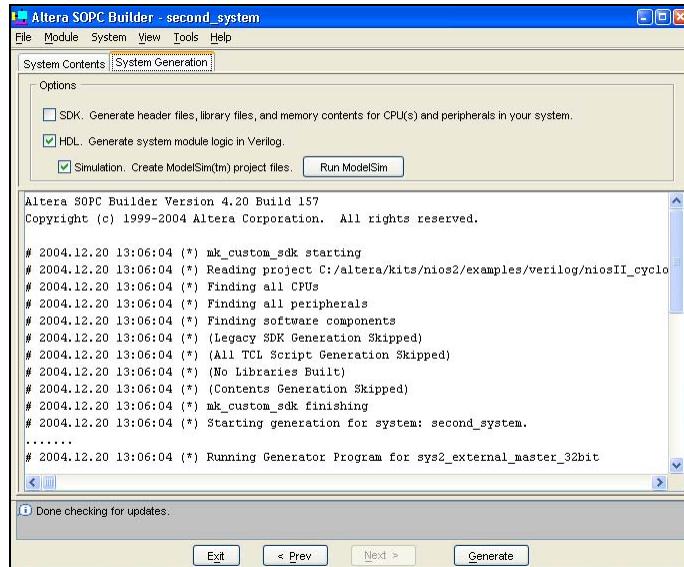


System Generation Tab

This section describes the settings available on the **System Generation** tab, and discusses the various outputs of SOPC Builder. System generation refers to SOPC Builder's process of generating output files that describe your system, based on the parameters you specified in the SOPC Builder GUI.

Figure 2–8 shows an example of the **System Generation** tab.

Figure 2–8. System Generation Page



Some SOPC Builder components, such as the Nios II processor, may modify the available options on this page.

Depending on the options on the **System Generation** tab, SOPC Builder generates the following outputs:

- Hardware design files
- Simulation model, testbench, and ModelSim project files
- Files for software development

System Generation Tab Options

This section describes each of the options available on the **System Generation** tab.

SDK Option

When the **SDK** option is turned on, during system generation SOPC Builder creates a custom software development kit (SDK) directory in the Quartus II project directory for each CPU in the system. The SDK contains a memory map and software files (drivers, libraries, and utilities) for any system components that provide software support files.



The Nios II processor provides a different software development flow, and therefore does not use the **SDK** option.

SDK files are arranged into the following directories:

- **inc** – This directory contains header files. These files include the definition for the memory map, register declarations for included peripherals, and macros that can be used in creating embedded software applications.
- **lib** – This directory contains software library files. During system generation, processor components can include commands to have SOPC Builder compile the libraries automatically.
- **src** – This directory provides a location for application source code development. Example source code files associated with peripherals may also be copied into this directory during system generation.

Every time you generate or update the system hardware module, you must give the SDK files to the software engineers developing application code.



If you edit the files generated by SOPC Builder, save them with a unique filename to prevent the file from being overwritten in a subsequent system generation.

HDL Option

When the **HDL** option is turned on, during system generation SOPC Builder creates HDL files that describe the system, and stores them in the Quartus II project directory. The HDL files contain the following:

- The top-level system module
- An instance of every component in the system
- The Avalon switch fabric tailored to connect all components in the system
- A simulation model and a simulation testbench, depending on the Simulation option. See “[Simulation Option](#)” on page [2-12](#) for more details.

SOPC Builder outputs the top-level system module file in either Verilog HDL or VHDL, depending on which language you specified when starting SOPC Builder. However, some SOPC Builder components may provide source files in only one language.

Simulation Option

When the **Simulation** option is turned on, during generation SOPC Builder creates a simulation model and a test bench for the system. Simulation-specific files are written to a simulation directory in the Quartus II project directory, separate from the synthesizable HDL files.

The testbench is tailored to the structure of system module. The testbench provides the following functionality:

- Instantiates the system module
- Drives clock and reset inputs with default behaviors
- Instantiates and connects the simulation models provided for any components external to the system module, such as memory models

Individual components may also provide simulation files, which SOPC Builder copies into the simulation directory during system generation.

SOPC Builder generates a ModelSim project directory that includes the following files:

- A ModelSim Project File (**.mpf**) for the current system
- Simulation data files for all memory components that have initialized contents
- A **setup_sim.do** file that contains setup information and aliases customized for simulating the system module
- A **wave_presets.do** file that automatically displays a default set of useful waveforms

You can run the ModelSim software directly from SOPC Builder by clicking **Run ModelSim**. This requires that the ModelSim software be specified in your search path in the **Setup** window (File menu).



Run ModelSim might be disabled by a component in the system. For example, Nios II processor systems provide a different simulation flow, and therefore **Run ModelSim** is unavailable on the **System Generation** tab for Nios II processor systems.

The SOPC Builder output files for simulation are designed to work with the ModelSim simulator. You can use the SOPC Builder-generated simulation model and testbench with other HDL simulators. While the

ModelSim-specific files (**.tcl**, **.do**, **.mpf**, etc.) will not work directly with other simulators, you can inspect these files and use them as a basis for setting up similar capabilities.

Starting System Generation

After you have configured all system parameters and specified the desired generation options on this tab, clicking **Generate** starts the system generation process. **Generate** is available from all tabs in SOPC Builder. It is disabled whenever there are any errors displayed in the messages window.

The middle area of the **System Generation** tab is a progress display, showing a time-stamped list of progress messages that occur during system generation. SOPC Builder executes multiple tools and scripts to generate the system, and the status messages from each step are reported in the progress display.

If system generation completes successfully, SOPC Builder displays a final progress message:

```
# <timestamp> SUCCESS: SYSTEM GENERATION COMPLETED.
```

SOPC Builder saves a log file of the progress messages in the Quartus II project directory.

Other Tools

The Tools menu provides access to a dynamic list of downstream design tools, depending on the components in the system. For example, if the system contains a Nios II processor, Nios II processor-related tools are listed in the Tools menu.

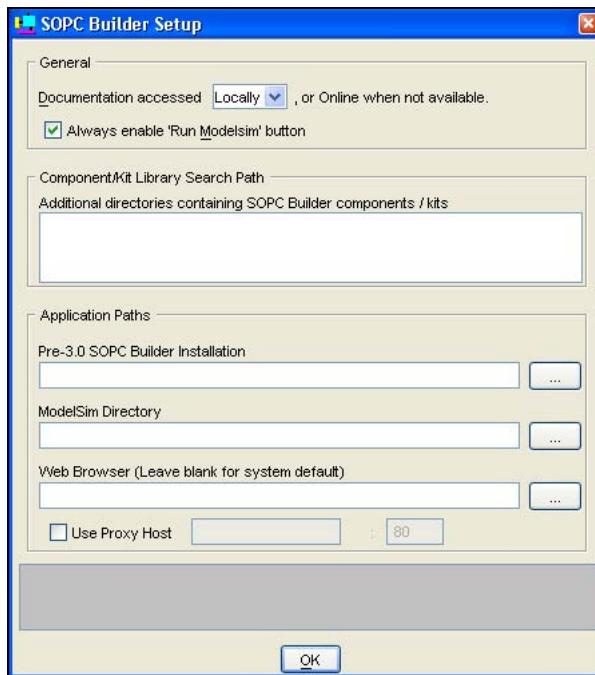
Preferences

The **SOPC Builder Setup** window (File menu) provides global options that affect SOPC Builder's operation, such as:

- Search path for SOPC Builder components
- Install path for ModelSim
- Location of documentation files
- Web browser settings (for accessing component documentation)

The **SOPC Builder Setup** window is shown in [Figure 2–9](#).

Figure 2–9. SOPC Builder Setup Window



Introduction

Avalon™ switch fabric is a high-bandwidth interconnect structure that consumes minimal logic resources, and provides greater flexibility than a typical shared system bus. This chapter describes the functions of Avalon switch fabric and the implementation of those functions.



For details on the Avalon interface, see the *Avalon Interface Specification* available at www.altera.com. For details on how to use SOPC Builder to create Avalon switch fabric, see the *Tour of the SOPC Builder User Interface* chapter in Volume 4 of the *Quartus II Handbook*.

High-Level Description

Avalon switch fabric is the glue that binds together components in a system based on the Avalon interface.

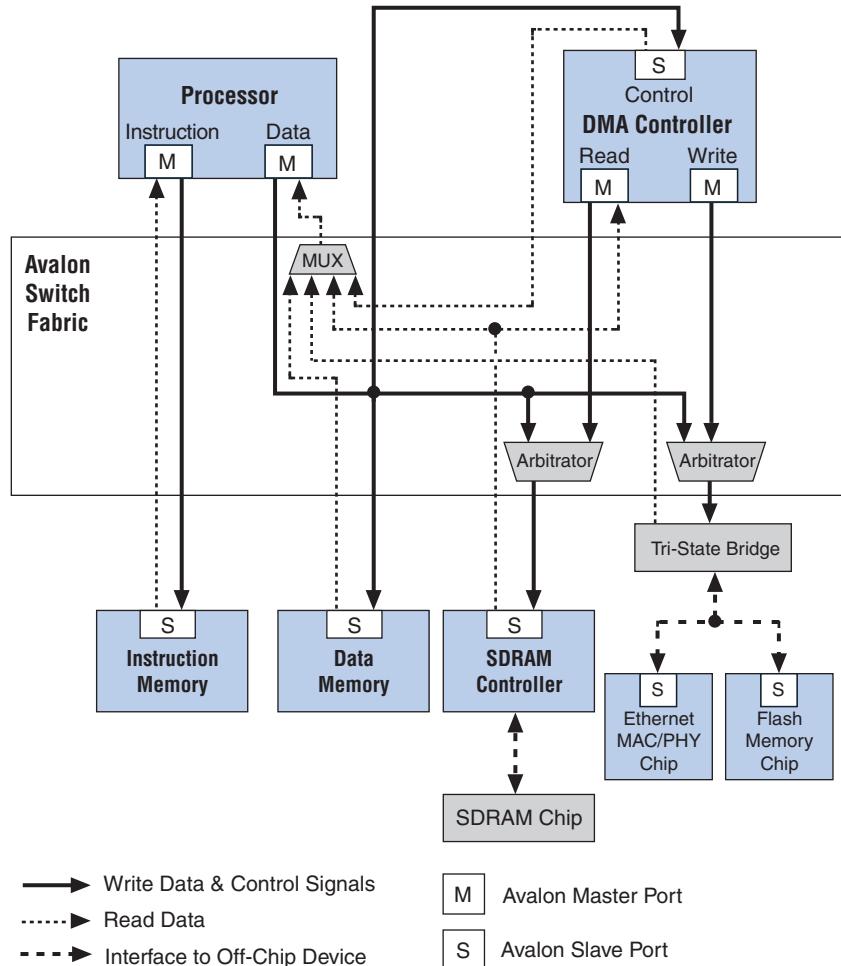
Avalon switch fabric is the collection of interconnect and logic resources that connects Avalon master and slave ports on components in a system. Avalon switch fabric encapsulates the connection details of a system. The Avalon switch fabric guarantees that signals travel correctly between master and slave ports, as long as the ports adhere to the rules of the Avalon interface specification. As a result, system designers can think at a higher level and focus on the parts of a system that add value, rather than worry about the interconnect.

Avalon switch fabric supports:

- Single-master or multiple-master systems with any number of slave components
- On-chip components
- Interfaces to off-chip peripherals
- Components of differing data widths
- Components operating in different clock domains
- Components using multiple Avalon ports

Figure 3–1 shows a simplified diagram of the Avalon switch fabric in an example system with multiple masters.

Figure 3–1. Avalon Switch Fabric Block Diagram – Example System



Some components in Figure 3–1 use multiple Avalon ports. Because an Avalon component can have multiple Avalon ports, you can use Avalon switch fabric to create *super interfaces* that provide more functionality than a single Avalon port. For example, an Avalon slave port can have only one interrupt-request (IRQ) signal. However, by using three Avalon slave

ports together, you can create a component that generates three separate IRQs. In this case, SOPC Builder generates the Avalon switch fabric to connect all ports.

Generating Avalon switch fabric is SOPC Builder's primary purpose. Because SOPC Builder generates Avalon switch fabric automatically, most users do not interact directly with it or the HDL that describes it. You do not need to know anything about the internal workings of Avalon switch fabric to take advantage of the services it provides. On the other hand, a basic understanding of how it works can help you optimize your components and systems to take the greatest advantage of the services it provides. For example, knowledge of the arbitration mechanism can help designers of multi-master systems minimize the impact of arbitration on the system throughput.

Fundamentals of Avalon Switch Fabric Implementation

Avalon switch fabric uses active logic to implement a switched interconnect structure that provides a dedicated path between master and slave ports. Avalon switch fabric consists of synchronous logic and routing resources inside an FPGA.

At each port interface, Avalon switch fabric manages Avalon transfers, responding to signals from the connected component. The signals that appear on the master port and corresponding slave port during a transfer can be very different, depending on how the Avalon switch fabric transports signals between the master-slave pair. In the path between master and slave ports, the Avalon switch fabric can introduce registers for timing synchronization, finite state machines for event sequencing, or nothing at all, depending on the services required by those ports.

Functions of Avalon Switch Fabric

Avalon switch fabric logic provides the following functions:

- Address Decoding (see page 3–4)
- Data-Path Multiplexing (see page 3–5)
- Wait-State Insertion (see page 3–6)
- Latency Capabilities (see page 3–7)
- Native Address Alignment & Dynamic Bus Sizing (see page 3–8)
- Arbitration for Multi-Master Systems (see page 3–11)
- Clock Domain Crossing (see page 3–18)
- Interrupt Controller (see page 3–21)
- Reset Distribution (see page 3–24)

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 the SOPC Builder GUI. The remaining sections of this chapter describe how SOPC Builder implements each function.

Address Decoding

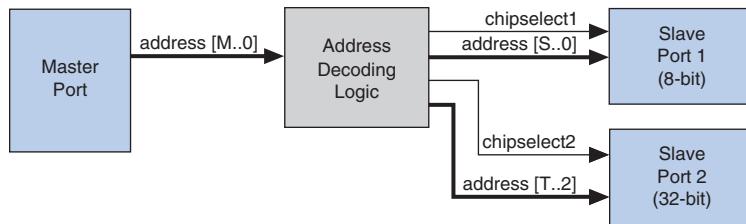
Address decoding logic in the Avalon switch fabric distributes an appropriate address, and produces a chipselect signal for each slave port. Address decoding logic provides the following benefits that simplify the design of the components:

- The Avalon switch 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 always properly aligned for the data width of 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 3–2 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 3–2, the address decoding logic handles the difference between the master address width (M) and the individual slave address widths (S & T). It also maps only the necessary master address bits to access words in each slave port's address space.

Figure 3–2. Block Diagram of Address Decoding Logic



All figures in this chapter are simplified to show only the particular function being discussed. In a complete system, the Avalon switch fabric might alter the address, data, and control paths beyond what is shown in any one particular figure.

In the SOPC Builder GUI, the user-configurable aspects of address decoding logic are controlled by the **Base** setting in the list of active components on the **System Contents** page, as shown in [Figure 3–3](#).

Figure 3–3. Base Settings in SOPC Builder Control Address Decoding

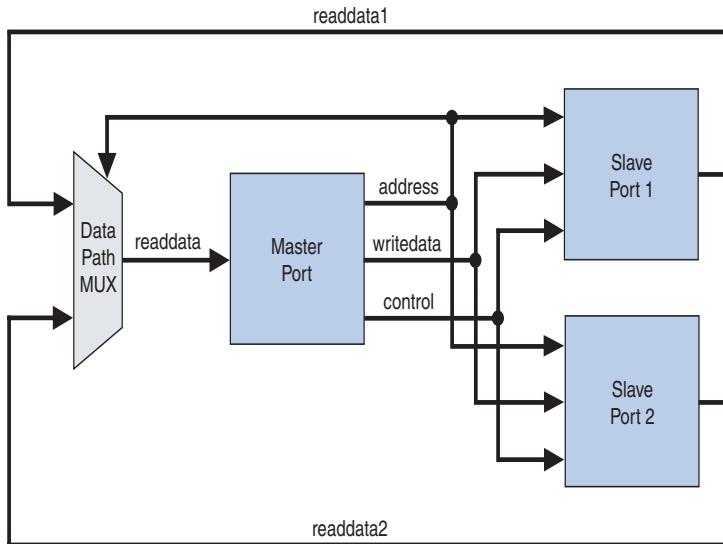
Module Name	Description	Base	End	IRQ
cpu	Nios II Proces...			
instruction_master	Master port			
data_master	Master port			
jtag_debug_mod...	Slave port	IRQ 0 0x02120000	IRQ 31 0x021207FF	
ext_flash	Flash Memory ...	0x00000000	0x007FFFFF	
ext_ram	IDT71V416 S...	0x20000000	0x20FFFFFF	
ext_ram_bus	Avalon Tri-St...			
button_pio	PIO (Parallel I/O)	0x02120860	0x0212086F	2
high_res_timer	Interval timer	0x02120820	0x0212083F	3

Data-Path Multiplexing

Data-path multiplexing logic in the Avalon switch fabric aggregates read-data signals from multiple slave ports during a read transfer, and presents the signals from only the selected slave back to the master port.

[Figure 3–4](#) shows a block diagram of the data-path multiplexing logic for one master and two slave ports. Separate data-path multiplexing logic is generated for every master port in the system.

Figure 3–4. Block Diagram of Data-Path Multiplexing Logic



Data-path multiplexing is not necessary in the write-data direction for write transfers. The `writedata` signals are distributed equally to all slave ports, and each slave port ignores `writedata` except for when the address-decoding logic selects that port.

In the SOPC Builder GUI, the generation of data-path multiplexing logic is specified using the connections panel on the **System Contents** page, as shown in [Figure 3–5](#).

Figure 3–5. Connection Panel Settings in SOPC Builder Control Data-Path Multiplexing

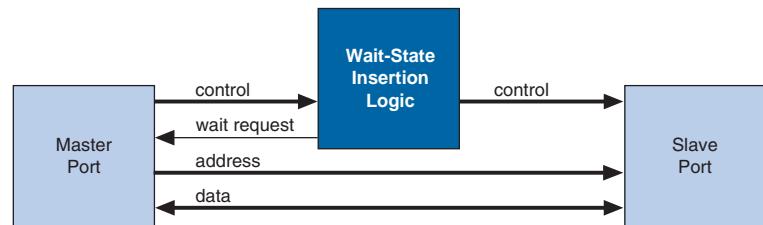
Use	Module Name	Description	Base	End	IRQ
<input checked="" type="checkbox"/>	cpu	Nios II Processor...			
	-instruction_...	Master port			
	-data_master	Master port			
	-tag_debug_...				
<input checked="" type="checkbox"/>	ext_ram_bus	Slave port	IRQ 0 0x02120000	IRQ 31 0x021207FF	
<input checked="" type="checkbox"/>	dram	Avalon Tri-State ...			
<input checked="" type="checkbox"/>	button_pio	SDRAM Controller	0x01000000	0x01FFFFFF	
<input checked="" type="checkbox"/>	-high_res_ti...	PIO (Parallel IO)	0x02120860	0x0212086F	2
<input checked="" type="checkbox"/>	jtag_uart	Interval timer	0x02120820	0x0212083F	3
<input checked="" type="checkbox"/>	lan91c111	JTAG UART	0x021208B0	0x021208B7	1
<input checked="" type="checkbox"/>		LAN91c111 Inter...	0x02110000	0x0211FFFF	6

Wait-State Insertion

Wait states extend the duration of a transfer by one or more cycles for the benefit of components with special synchronization needs. Wait-state insertion logic accommodates the timing needs of each slave port, and coordinates the master port to wait until the slave can proceed. The Avalon switch fabric inserts wait states into a transfer when the target slave port cannot respond in a single clock cycle. Avalon switch 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 3–6](#) shows a block diagram of the wait-state insertion logic between one master and one slave.

Figure 3–6. Block Diagram of Wait-State Insertion Logic



The Avalon switch 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 multi-master 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.

Latency Capabilities

Latency-aware components are capable of pipelining read transfers, starting multiple read transfers in succession without waiting for the prior transfers to complete. Transfers with latency allow master-slave pairs to achieve maximum throughput, even though the slave port may require one or more cycles of latency to return data for each transfer.

SOPC Builder generates the Avalon switch fabric to take advantage of latency-aware components wherever possible, based on the latency properties of each master-slave pair in the system. Regardless of the latency of a target slave port, SOPC Builder guarantees that data comes back to each master in the order requested. Because master and slave

ports often have mismatched latency capabilities, the Avalon switch fabric often contains logic to reconcile the differences. Many cases are possible, as shown in [Table 3–1](#).

<i>Table 3–1. Various Cases of Latency in a Master-Slave Pair</i>		
Master Latency	Slave Latency	Avalon Switch Fabric Logic Structure
None	None	The Avalon switch fabric does not instantiate logic to handle latency.
None	Fixed or Variable	The Avalon switch fabric forces the master port to wait through any slave-side latency cycles. This master-slave pair gains no benefits of latency, because the master port is not latency aware 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.
Capable	None	The Avalon switch fabric carries out the transfer as if neither port were latency aware, forcing the master port to wait until the slave port returns data.
Capable	Fixed	The Avalon switch 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.
Capable	Variable	This is the simplest latency case, in which the slave port asserts a signal when its data 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 latency based on the properties of the master and slave ports in the system. When configuring a system in SOPC Builder, there are no GUI settings that directly control the latency logic in the Avalon switch fabric.

Native Address Alignment & Dynamic Bus Sizing

SOPC Builder generates Avalon switch 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 it may 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 3–2 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 0x00000000 in the master port. In **Table 3–2**, OFFSET refers to the offset into the 16-bit slave port address space.

Table 3–2. 32-Bit Master View of 16-Bit Slave Data		
32-bit Master Address	Data with Native Alignment	Data with Dynamic Bus Sizing
0x00000000 (word 0)	0x0000:OFFSET [0]	0xOFFSET [1] :OFFSET [0]
0x00000004 (word 1)	0x0000:OFFSET [1]	0xOFFSET [3] :OFFSET [2]
0x00000008 (word 2)	0x0000:OFFSET [2]	0xOFFSET [5] :OFFSET [4]
0x0000000C (word 3)	0x0000:OFFSET [3]	0xOFFSET [7] :OFFSET [6]
...		
(word N)	0x0000:OFFSET [N]	0xOFFSET [2 N +1] :OFFSET [2 N]



The Avalon interface is little endian.

SOPC Builder generates address-alignment logic appropriately based on the properties of the master and slave ports in the system. When configuring a system in SOPC Builder, there are no GUI settings that directly control the address alignment in the Avalon switch fabric.

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 (i.e. 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, the Avalon switch fabric maps all slave data bits to the lower bits of the master data, and fills any remaining upper bits with zero. The Avalon switch fabric performs simple wire-mapping in the data path, 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.

Dynamic Bus Sizing

Slave ports that access memory devices generally use dynamic bus sizing. Dynamic bus sizing hides the details of interfacing a narrow memory 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 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 seamlessly, regardless of the data width.

In the case of dynamic bus sizing, the Avalon switch 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 N slave transfers. The master port pays a performance penalty, because it must wait while multiple slave-side transfers complete.

In the case of a read transfer, the Avalon switch fabric concatenates multiple units of slave data and presents them to the master port as a single unit of data. The master port is forced to wait until all slave-side read transfers have completed. A read transfer from a wide master port always causes multiple slave read transfers to sequential addresses in the slave's address space. For example, even if a 32-bit master port needs only one byte from a dynamically-aligned 8-bit memory, a master read transfer will generate four slave transfers, and the master port will wait until all four transfers complete.

During write transfers, the dynamic bus sizing logic uses the master-side byte-enable signals to generate appropriate slave write transfers. The dynamic bus sizing logic will perform as many slave-side transfers as necessary to write the specified byte lanes to the slave memory.

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. There is no performance penalty when accessing a wider slave port using dynamic bus sizing.

Table 3–3 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 3–3. 32-Bit Master View of 64-Bit Slave with Dynamic Bus Sizing

32-bit Address	Data
0x00000000 (word 0)	OFFSET [0] _{31..0}
0x00000004 (word 1)	OFFSET [0] _{63..32}
0x00000008 (word 2)	OFFSET [1] _{31..0}
0x0000000C (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 the slave-side byte-enable signals to write only to the appropriate byte lanes.

Arbitration for Multi-Master Systems

Avalon switch fabric supports systems with multiple master components. In a system with multiple master ports, such as the system pictured in [Figure 3–1 on page 3–2](#), the Avalon switch 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 multi-master architecture used by Avalon switch 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 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 the SOPC Builder GUI. 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 switch fabric. Each Avalon master port connects to the Avalon switch fabric like it is the only master port in the system. As a result, you can reuse a component in single-master and multi-master systems without requiring design changes to the component.

This section discusses the architecture of the Avalon switch fabric generated by SOPC Builder for multi-master systems.

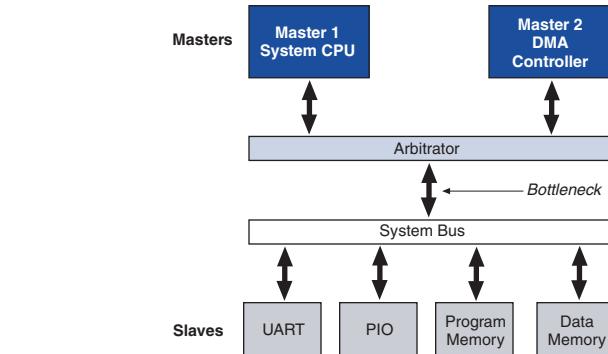
Traditional Shared Bus Architectures

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 arbitrator controls the bus (i.e., the path between bus masters and bus slaves), so that multiple bus masters do not simultaneously drive the bus and cause electrical contention. Each bus master requests control of the bus from the arbitrator, and the arbitrator 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 arbitrator 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 arbitrator always grants control to the master with the highest priority—is used in many existing bus architectures.

Figure 3–7 illustrates the bus architecture for a traditional processor system. Access to the shared system bus becomes the bottleneck for throughput and utilization performance. Only one master has access to the bus at a time, which means that other masters are forced to wait (diminishing throughput), and only one slave can transfer data at a time (diminishing utilization).

Figure 3–7. Bus Architecture in a Traditional Microprocessor System



Slave-Side Arbitration

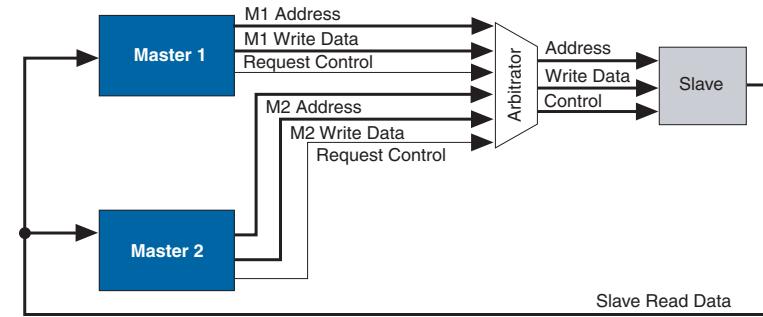
The multi-master architecture used by Avalon switch fabric eliminates the bottleneck for access to a shared bus, because the system does not have shared bus lines. Avalon master-slave pairs are connected by dedicated paths. A master port never waits to access a slave port, unless a different master port 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 multi-master Avalon system still 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 3–1 on page 3–2](#) demonstrates a system with two master ports (a CPU and a DMA controller) sharing a slave port (an SDRAM controller). Arbitration is performed on the SDRAM slave port; the arbitrator dictates which master port gains access to the slave port if both master ports initiate a transfer with the slave port at the same time.

Figure 3–8 focuses on the two master ports and the shared slave port, and shows additional detail of the data, address, and control paths. The arbitrator logic multiplexes all address, data, and control signals from a master port to a shared slave port.

Figure 3–8. Detailed View of Multi-Master Connections



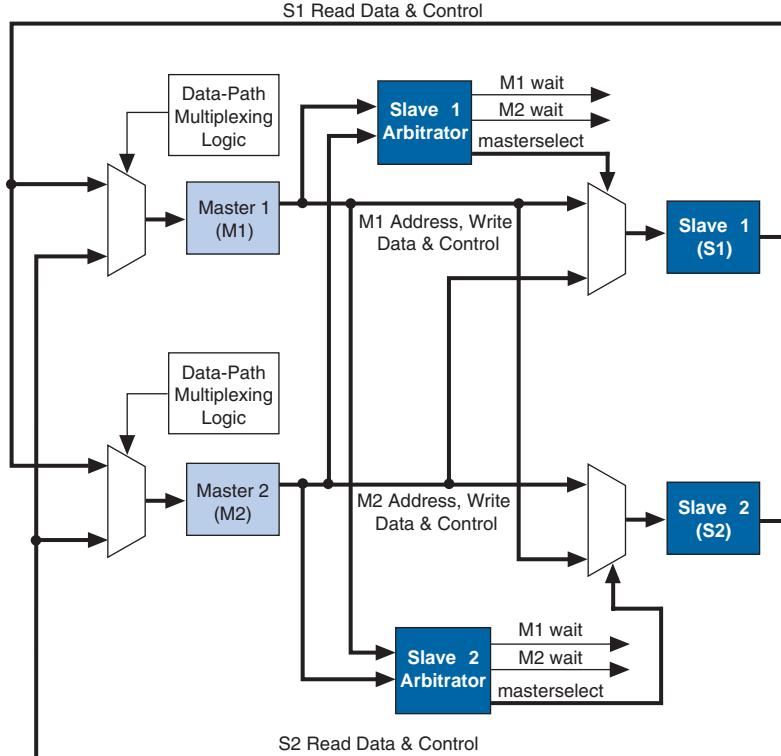
Arbitrator Details

SOPC Builder generates an arbitrator for every slave port connected to multiple master ports, based on arbitration parameters specified in the SOPC Builder GUI. The arbitrator logic performs the following functions for its associated slave port:

- Evaluates the address and control signals from each master port at every clock cycle when a new transfer can begin, and determines which master port (if any) is requesting access to the slave.
- Chooses which master port gains access to the slave next.
- Grants access to the chosen master port (i.e. allows it to proceed with the transfer), and forces all other requesting master ports to wait.
- Uses multiplexers to connect address, control, and data paths between the multiple master ports and the slave port. The arbitrator logic guarantees that an appropriate master port (if any) is connected to the slave port.

Figure 3–9 shows the arbitrator logic in an example multi-master system with two master ports, each connected to two slave ports.

Figure 3–9. Block Diagram of Arbitrator Logic



Arbitration Rules

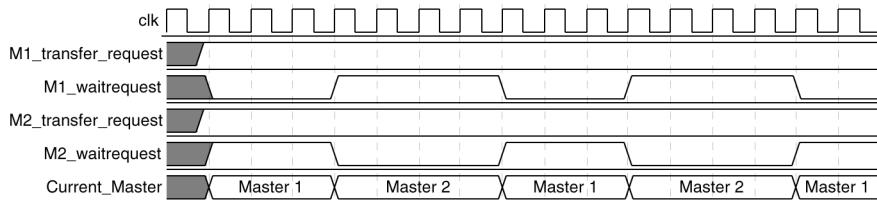
This section describes the rules by which the arbitrator grants access to master ports when they contend.

Fairness-Based Shares

The Avalon arbitrator 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 is permission to perform one transfer.

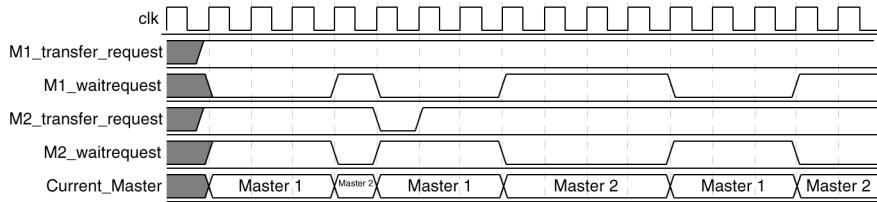
For example, assume that two master ports continuously attempt to perform back-to-back transfers to a slave port. Master 1 is assigned three shares and Master 2 is assigned four shares. In this case, the arbitrator will grant Master 1 access for three transfers, then Master 2 for four transfers. This cycle will repeat indefinitely. [Figure 3–10](#) demonstrates this case, showing each master port's transfer request output, wait request input (which is driven by the arbitrator logic), and the current master with control of the slave.

Figure 3–10. 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 arbitrator grants access to another requesting master. See [Figure 3–11 on page 3–16](#). After completing one transfer, Master 2 stops requesting for one clock cycle. As a result, the arbitrator grants access back to Master 1, which gets a replenished supply of shares.

Figure 3–11. 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 arbitrator grants shares in round-robin order. At every slave transfer, only requesting master ports are included in the round-robin arbitration.

Minimum Share Value

A component design can declare the minimum number of shares in each round-robin cycle, which affects how the arbitrator grants access. For example, if a slave port has a minimum share value of ten, then the arbitrator will grant at least ten shares to any master port when it begins a sequence of transfer requests. The arbitrator might grant more shares, if the master port is assigned more shares in the SOPC Builder GUI.

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.



See the *Avalon Interface Specification* for details on burst transfers. Burst transfers provide even higher performance for continuous transfers when they are guaranteed to be sequential.

Setting Arbitration Parameters in the SOPC Builder GUI

You specify the arbitration shares for each master using the connection panel on the **System Contents** tab of the SOPC Builder GUI, as shown in Figure 3–12.

Figure 3–12. 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 → jtag_debug_module	Slave port	
1 sys_clk_timer	Interval timer	clk
1 ext_ram_bus	Avalon Tri-State Bridge	clk
ext_flash	Flash Memory (Commo...	
ext_ram	IDT71V416 SRAM	
1 epc_controller	EPCS Serial Flash Cont...	clk
lan91c111	LAN91c111 Interface (...)	
1 jtag_uart	JTAG UART	clk



The arbitration settings are hidden by default. To view them, choose **Show Arbitration** (View menu).

Clock Domain Crossing

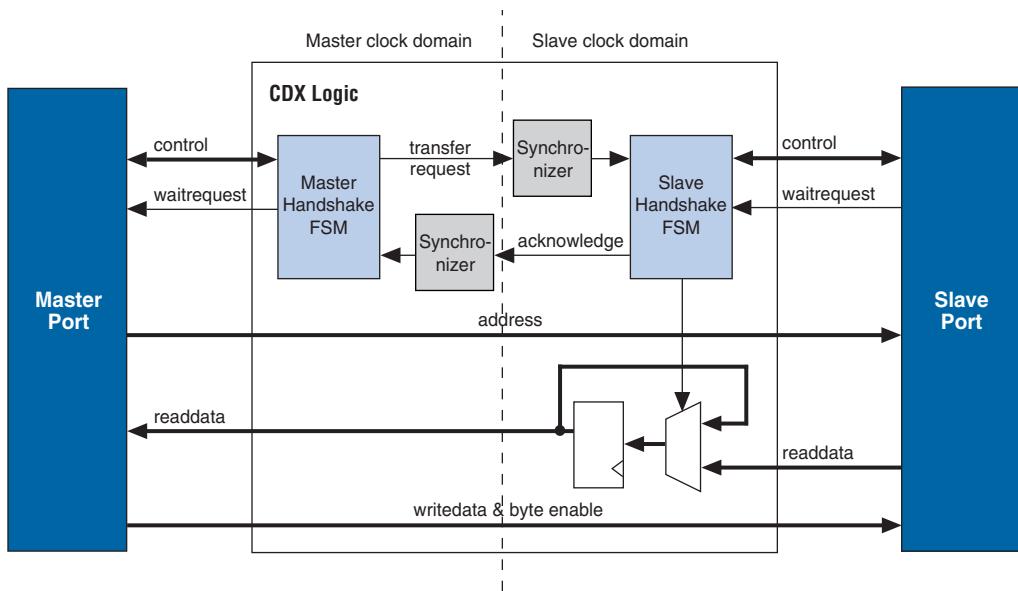
SOPC Builder generates clock-domain crossing (CDX) logic that hides the details of interfacing components operating in asynchronous clock domains. The Avalon switch fabric upholds the Avalon protocol with each port independently, and therefore each Avalon port need only be aware of its own clock domain. The Avalon switch fabric logic propagates transfers across clock domain boundaries transparently to the user.

The CDX logic in Avalon switch 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 CDX hardware manually.
- Each Avalon port operates in only one clock domain, which reduces design complexity of components.
- Enables master ports to access any slave port without awareness of 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 CDX logic consists of two finite state machines (FSM), one in each clock domain, which 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 3–13 on page 3–19](#) shows a block diagram of the clock domain crossing logic between one master and one slave port.

Figure 3–13. Block Diagram of Clock Domain-Crossing Logic

The Synchronizer blocks in Figure 3–13 use multiple stages of flip-flops to eliminate the propagation of metastable events on the control signals that enter the hand-shake FSMs.

The CDX logic works with any clock ratio. Altera® tests the CDX 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 CDX 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 slave perspective, there is nothing different about a transfer initiated by a master in a different clock domain. From the master perspective, a transfer across clock domains simply takes extra clock cycles. Similar to other transfer delay cases (for example, arbitration delay and/or wait states on the slave side), the Avalon switch 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 CDX logic, based on the system contents and the connections between components. SOPC Builder places CDX logic to maintain the highest transfer rate for all components. SOPC Builder generates clock domain crossing logic on a slave-by-slave basis; every slave port can be considered independently as its own clock domain crossing exercise.

Duration of Transfers Crossing Clock Domains

CDX 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

Implementing Multiple Clock Domains in the SOPC Builder GUI

You specify the clock domains used by your system on the **System Contents** tab of the SOPC Builder GUI. You define the input clocks to the system using the **Clock** settings, shown in [Figure 3–14](#). Clock domains are differentiated based on the name of the clock. It is possible to create multiple asynchronous clocks with the same frequency.

Figure 3–14. Clock Settings on the System Contents Tab



After you define the system clocks, you specify which clock drives which components using the table of active components, as shown in [Figure 3–15](#).

Figure 3–15. Assigning Clocks to Components

Module Name	Description	Clock	Base	End	IRQ
lcd_display	Character LCD (16x2, ...)	clk	0x02120820	0x0212083F	3
high_res_timer	Interval timer	clk	0x02120890	0x0212089F	
seven_seg_pio	PIO (Parallel I/O)	clk	0x021208A0	0x021208AF	
reconfig_request_pio	PIO (Parallel I/O)	fastclk			
uart1	UART (RS-232 serial port)	clk	0x02120840	0x0212085F	4
sysid	System ID Peripheral	clk	0x02120888	0x021208BF	
sdram	SDRAM Controller	clk	0x01000000	0x0FFFFFFF	
dma_0	DMA	fastclk	0x00800000	0x008001FF	7
read_buffer	On-Chip Memory (RAM ...)	fastclk	0x00801000	0x00801FFF	
write_buffer	On-Chip Memory (RAM ...)	fastclk	0x00802000	0x00802FFF	

For further details, see the *Building Systems with Multiple Clock Domains* chapter in Volume 4 of the *Quartus II Handbook*.

Interrupt Controller

In systems with one or more slave ports that generate IRQs, the Avalon switch 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.

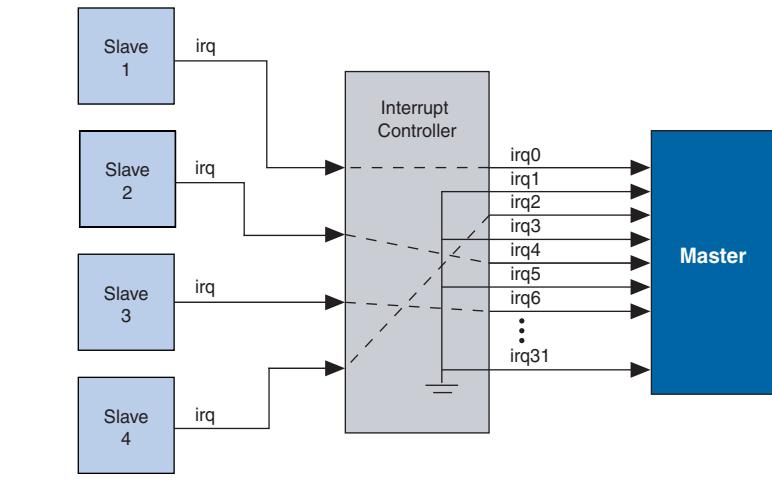
Each slave port optionally produces an IRQ output signal. There are two master signals related to interrupts: `irq` and `irqnumber`. SOPC Builder generates the interrupt controller in one of two configurations, software priority or hardware priority, depending on the interrupt signals present on the master port.

Software Priority

In the software priority configuration, the Avalon switch 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 3–16](#) shows an example of the interrupt controller mapping the IRQs on four slave ports to `irq[31..0]` on a master port.

Figure 3–16. IRQ Mapping Using Software Priority

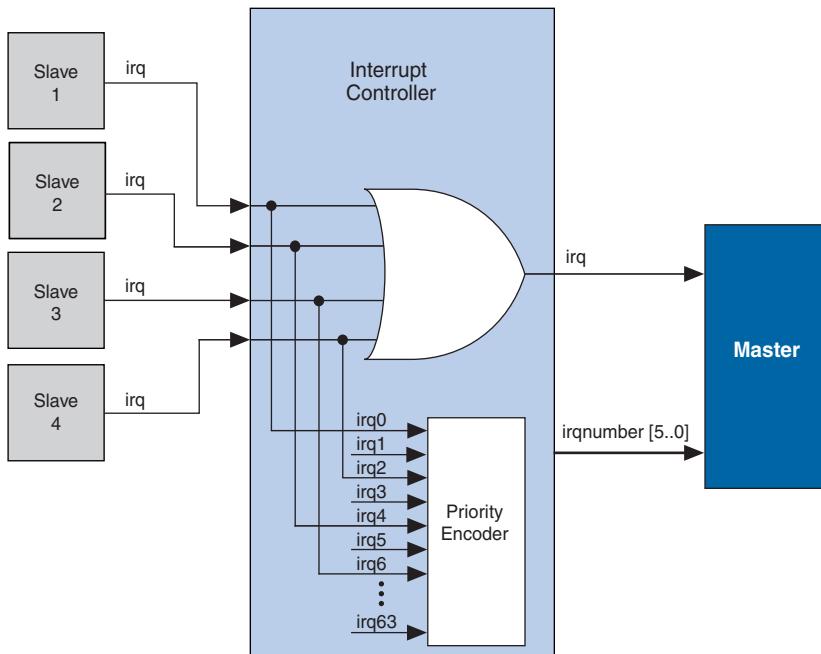


Hardware Priority

In the hardware priority configuration, in the event that multiple slaves assert their IRQs simultaneously, the Avalon switch fabric (i.e. 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 3-17](#).

Figure 3-17. IRQ Mapping Using Hardware Priority



Assigning IRQs in the SOPC Builder GUI

You specify IRQ settings on the **System Contents** tab of the SOPC Builder GUI. 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 3–18 shows the IRQ settings for multiple slave IRQs that drive the master component named **cpu**.

Figure 3–18. Assigning IRQs in the SOPC Builder GUI

Module Name	Description	Clock	Base	End	IRQ
cpu	Nios II Processor - Alter...	clk	0x02120000	0x021207FF	↑
ext_ram_bus	Avalon Tri-State Bridge	clk			
ext_flash	Flash Memory (Common ...)		0x00000000	0x007FFFFF	
ext_ram	IDT71V416 SRAM		0x02000000	0x020FFFFF	
epcs_controller	EPCS Serial Flash Contr...	clk	0x02100000	0x021007FF	NC
lan91c111	LAN91c111 Interface (E...		0x02110000	0x0211FFFF	6
sys_clk_timer	Interval timer	clk	0x02120800	0x0212081F	↓
jtag_uart	JTAG UART	clk	0x021208B0	0x021208B7	1
button_pio	PIO (Parallel I/O)	clk	0x02120860	0x0212086F	2
led_pio	PIO (Parallel I/O)	clk	0x02120870	0x0212087F	3
high_res_timer	Interval timer	clk	0x02120820	0x0212083F	
lcd_display	Character LCD (16x2, O...	clk	0x02120880	0x0212088F	
analog_sar_pio	DIO (Parallel I/O)	all	0x02120890	0x0212089F	

Reset Distribution

The Avalon switch fabric generates and distributes a system-wide reset pulse to all logic in the system module. The switch fabric distributes the reset signal conditioned for each clock domain. The duration of the reset signal is at least one clock period.

The Avalon switch fabric asserts the system-wide reset in the following conditions:

- The global reset input to the system module has been asserted.
- A slave port has asserted its `resetrequest` signal.
- The FPGA has been reconfigured.

All components must enter a well-defined reset state whenever the Avalon switch fabric asserts the system-wide reset. The timing of the reset signal is asynchronous to the operation of transfers.

Introduction

This chapter describes in detail what an SOPC Builder component is. SOPC Builder components are individual design block that SOPC Builder uses to integrate a larger system module. Each component consists of a structured set of files within a directory.

The files in a component directory serve the following purposes:

- Defines the hardware interface to the component, such as the names and types of I/O signals.
- Declares any parameters that specify the structure of the component logic and the component interface.
- Describes a configuration wizard GUI for configuring the component in SOPC Builder.
- Provides scripts and other information SOPC Builder needs to generate the component HDL and integrate the component into the system module.
- Contains component-related information, such as software drivers, necessary for development steps downstream from SOPC Builder.



For details on creating custom components, see the *Developing SOPC Builder Components* chapter in Volume 4 of the *Quartus II Handbook*. For details on the SOPC Builder component editor, see the *Component Editor* chapter in Volume 4 of the *Quartus II Handbook*.

Sources of Components

There are several sources for components, including the following:

- The Quartus® II software, which includes SOPC Builder, installs a number of components.
- Altera® development kits, such as the Nios® II development kit, provide SOPC Builder components as features of the kit.
- Third-party developers provide SOPC Builder Ready components, including component directories and documentation on how to use the component.
- You can package your own HDL files into a new, custom component, using the SOPC Builder component editor.



While it is possible to write component files manually, Altera strongly recommends you use the SOPC Builder component editor to create custom components, for reasons of consistency and forward compatibility.

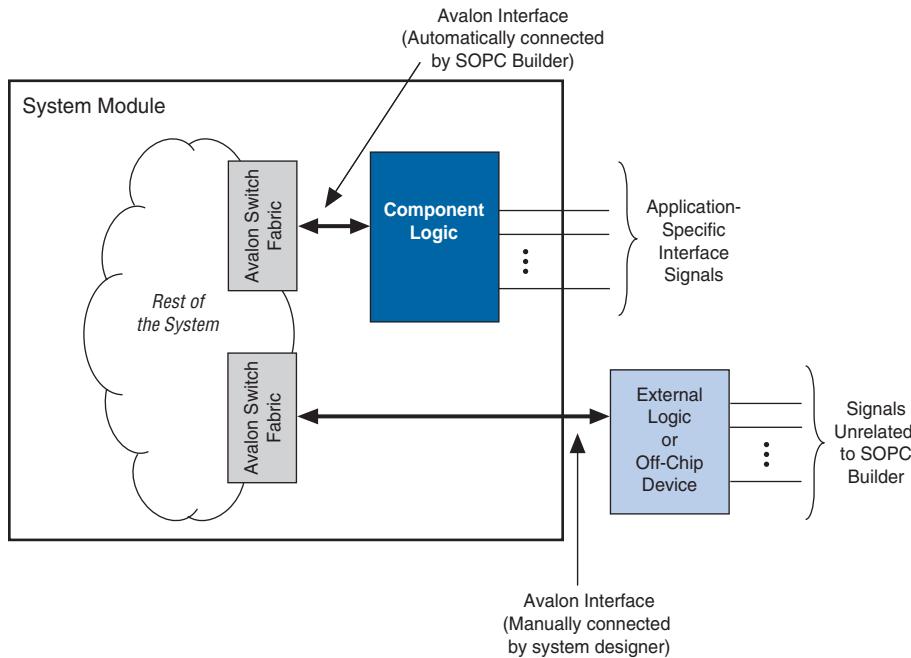
Location of the Component Hardware

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 component.

Figure 4–1. Component Logic Inside and Outside the System Module



Components That Include Logic Inside the System Module

In this case, the component files provide a full description of the component hardware. During system generation, SOPC Builder instantiates the component logic inside the system module and automatically wires the component to the rest of the system. Internal to the system module, the component connects to the rest of the system through its Avalon interface. The component can also have non-Avalon signals that SOPC Builder exposes on the top-level system module.

Components That Interface to Logic Outside the System Module

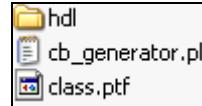
In this case, the component files describe only the interface to logic external to the system module. During system generation, SOPC Builder does not instantiate any logic for this component. Instead, SOPC Builder exposes an Avalon interface for this component on the top-level system module. You must manually wire the interface to external logic, such as a separate HDL module or an off-chip device.

As shown in Figure A, in this case there is no physical embodiment of the SOPC Builder component. Components that interface to external logic describe only the shape of the Avalon interface; they do not include logic inside the system module.

Structure & Contents of a Component Directory

This section describes the files that exist in a component directory. [Figure 4–2](#) shows a typical component directory created by the component editor.

Figure 4–2. Typical Component Directory



At a minimum, a component consists of a directory containing a file named **class.ptf**. Usually there are other files as well. The name of the enclosing directory is ignored. The **class.ptf** file defines everything that SOPC Builder needs to know about the name and location of component files.

As shown in [Figure 4–2](#), the component directory can contain the following items, and other files:

- **class.ptf** file
- **cb_generator.pl** script
- **hdl** directory

class.ptf File

SOPC Builder reads this file to find everything it needs to know about the component. The **class.ptf** file defines the following aspects of a component:

1. The component identity – The component **class.ptf** file defines the following properties that identify a component:
 - *Name* – The user-visible name of the component.
 - *Class* – The unique identifier that SOPC Builder uses to differentiate components. The class name uses only filename-friendly characters. When you instantiate a component in SOPC Builder, the instantiation name defaults to the class name.
 - *Version* – Identifies the version of a component. During system generation, SOPC Builder records the version number of each component in the system module.
 - *Group* – Determines in which group the component appears in the list of available components in the SOPC Builder GUI.
2. How the component hardware connects into a system module, including:
 - *Interface to the component* – The **class.ptf** file describes the blackbox structure of the component, defining the name, type, and direction of each interface signal.
 - *Method to generate a hardware instance of the component* – If the component includes logic inside the system module, the **class.ptf** file specifies a mechanism for SOPC Builder to generate the component hardware. The **class.ptf** file typically specifies a separate executable file to generate the component instance.
3. How the component presents its configurable options to the user – The **class.ptf** file declares the user-configurable parameters, and specifies a GUI for configuring those parameters.

cb_generator.pl File

This file is a Perl script used by SOPC Builder during system generation. Based on parameters you specify in the SOPC Builder GUI, the script generates one or more instances of this component to be instantiated in the top-level system module. The name of the file is not significant. The file name is defined in the **class.ptf** file.

For components generated by the SOPC Builder component editor, this script creates a Verilog or VHDL wrapper that instantiates HDL modules defined by files in the **hdl** directory. During system generation, a typical generator script copies the HDL files to the Quartus II project directory, and renames the files uniquely to match the instance name. The generator script can also define parameters for the top-level HDL module, based on parameters specified in the SOPC Builder GUI.



The generator script for most Altera-provided components dynamically generates HDL for each instance of the component, based on parameters specified in the SOPC Builder GUI. Such components do not include HDL files, because the HDL is generated programmatically.

hdl Directory

This subdirectory contains HDL files that describes the component hardware. This directory is not required to exist, and the name of the directory is not significant. The generator script specifies the exact path to HDL files, if any.

Other Component Files

The component directory can contain other files and subdirectories, depending on the component's design and recommended usage. Typically, such items are not used directly by SOPC Builder, but are necessary for other stages of development. SOPC Builder ignores any files it cannot identify for its own purposes.

Items you might find in a component directory include the following:

- **inc** subdirectory – This directory includes the register map for a peripheral component. The register map is typically declared as a C header file that defines symbols and macros for accessing the component hardware. SOPC Builder does not use this information directly. For Nios II processor systems, the Nios II IDE uses the contents of **inc**.
- **HAL** subdirectory – This directory contains software drivers for a component. SOPC Builder does not use this information directly. For Nios II processor systems, the Nios II IDE uses the contents of **HAL** to generate software for the component.
- **UCOSII** subdirectory – This directory contains software drivers for a component, specific to the MicroC/OS-II real-time operating system (RTOS). SOPC Builder does not use this information directly. For Nios II processor systems, the Nios II IDE uses the contents of **UCOSII** to generate software for the component.
- Files associated with the generator script – The generator script may require data files, Perl libraries, or other files.
- **sdk** subdirectory – This directory contains software and header files to support the legacy software development kit (SDK) flow for the first-generation Nios processor.
- Data files, source code, or other files needed by tools other than SOPC Builder

Component Directory Location

Each time SOPC Builder starts up, it looks for component directories. Any components that it finds are displayed in the list of available components on the SOPC Builder **System Contents** tab.

SOPC Builder searches the following locations for component directories:

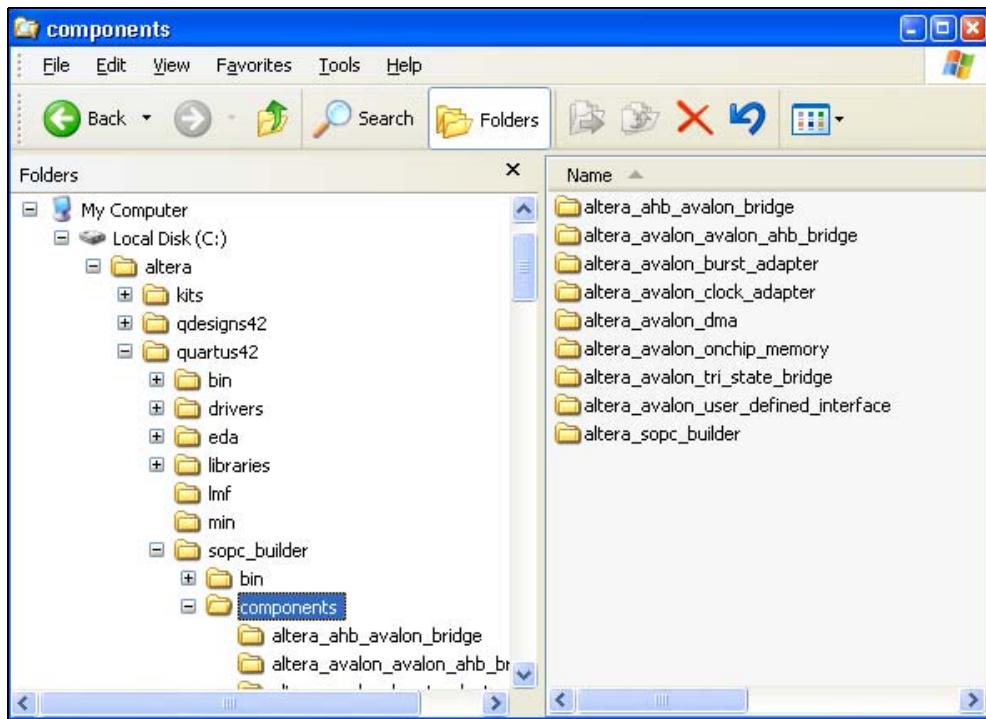
1. The current Quartus II project directory
2. Any directories specified under **Component/Kit Library Search Path** on the SOPC Builder Setup page in the SOPC Builder GUI.
3. **SOPC_BUILDER_PATH**
4. **QUARTUS_ROOTDIR/sopc_builder/**

The following describes the process by which SOPC Builder identifies components:

- **SOPC_BUILDER_PATH** and **QUARTUS_ROOTDIR** are environment variables that are defined during the Quartus II installation process.
- SOPC Builder searches each of these locations for subdirectories in the order shown, and then searches each subdirectory for a **class.ptf** file.
- When SOPC Builder finds a **class.ptf** file, it reads the file, identifies the component, and makes the component available in the SOPC Builder GUI.
- If multiple instances of the same component class exist, SOPC Builder uses the following rules to determine which one to use:
 - The component with the highest version takes precedence.
 - If all component versions are equal, the first component found takes precedence.
- If a directory is recognized as a kit directory (indicated by the presence of a file named **.sopc_builder**), then SOPC Builder further searches in the **components/** subdirectory.

Figure 4–3 shows an example of the directory of components installed with the Quartus II software. Note that not all directories correspond directly to user-visible components.

Figure 4–3. Avalon Component Directories



QI154005-1.0

Introduction

This chapter describes the SOPC Builder component editor. The component editor is a feature of SOPC Builder that lets you create and edit your own SOPC Builder components. You use the component editor graphical user interface (GUI) to do the following:

- Import hardware description language (HDL) files (if any) that describe the component hardware.
- Specify the hardware interface(s) to the component.
- Package software drivers into the component directory.
- Declare any parameters that alter the component structure or functionality, and define a user interface to let users parameterize instances of the component.

A typical development sequence might include the following steps, not necessarily in this order:

1. Design and test custom hardware in Verilog or VHDL
2. Build an SOPC Builder system
3. Use the component editor to package the custom HDL into an SOPC Builder component
4. Incorporate one or more instances of your custom component into the SOPC Builder system.
5. Test the system including the new component, and make iterative refinements to the HDL.
6. Use component editor to update the component files.

You can also use the component editor to define an Avalon™ interface to logic external to the system module. In this case, you do not provide HDL files, and use the component editor only to define the interface.

Component Editor Output

Based on the settings you make using the component editor GUI, the component editor outputs a properly-formed component directory, containing the following items:

- **class.ptf** file – This file identifies the component, defines how the component hardware connects to the overall system, and declares user-configurable parameters.
- **cb_generator.pl** file – This file is a script used by SOPC Builder during system generation to generate instances of the component hardware.
- **hdl** directory – If the component includes logic inside the system module, this directory contains the HDL files.
- Software files – If this component is a processor peripheral, you can associate software files, such as drivers.

Exiting the component editor automatically prompts you to save the component files. You can also choose **Save** (File menu) to save the files. Note that "saving" actually creates or copies multiple files, and stores them in their appropriate places.

The component editor always saves your new component to a subdirectory in the current Quartus® II project directory. You can relocate the component directory later, if you wish. For example, you could locate the component files in a central location so that other users can instantiate the component in their systems.

The component editor creates components with the following hardware characteristics:

- A component has one or more interfaces. Typically, an *interface* means an Avalon master port or slave port. The component editor lets you build a component with any combination of Avalon master or slave ports.
- Each interface is comprised of one or more signals.
- The component can use a limited set of global signals that are not associated to a specific Avalon interface.
- The component can include logic inside the system module, or serve as an interface to logic external to the system module.



See the *SOPC Builder Components* chapter in Volume 4 of the *Quartus II Handbook* for details on components. See the *Avalon Interface Specification* for details on the Avalon interface.

Starting the Component Editor

To start the component editor, choose **New Component** (File menu) in the SOPC Builder GUI. When the component editor starts, it displays the **Introduction** tab, which provides a simple introduction to using the component editor.

The component editor GUI presents several tabs that group like settings on the same tab. A message window at the bottom of the component editor displays warning and error messages.



Each tab in the component editor GUI provides on-screen information that describes how to use each tab. Click the triangle at the top-left of each tab to view these instructions.

In general, you will proceed through the tabs from left to right as you progress through the component creation process. You can return to an earlier tab at any time.

HDL Files Tab

You use the **HDL Files** tab to import existing Verilog or VHDL files that describe the component hardware. When you save the component later, the component editor copies these files into your new component's directory.

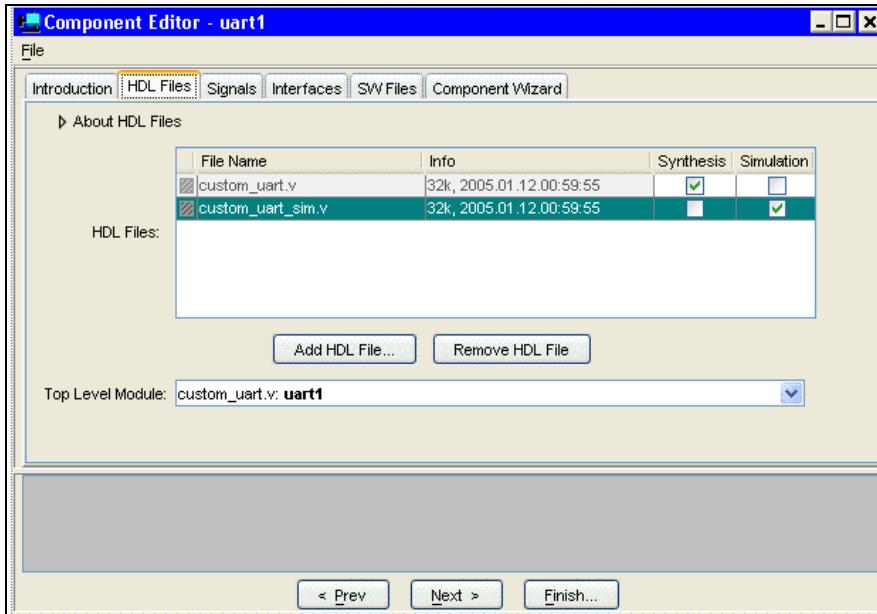


If your component is an interface to external logic only, then do not specify any files on this tab.

As you add each HDL file, the component editor analyzes the file by invoking the Quartus II Analyzer in the background. If there are errors, a dialog appears on the screen describing the problems. If the file is successfully analyzed, then the component editor identifies any design modules described in the file, and lists them in the **Top Level Module** list. If your HDL contains more than one module, you must select the appropriate top level module in the **Top Level Module** list.

Figure 5–1 shows an example of the **HDL Files** tab.

Figure 5–1. HDL Files Tab



If separate HDL files define the synthesizable hardware and simulation model for the component, use the **Synthesis** and **Simulation** boxes to specify the role for each file.

Signals Tab

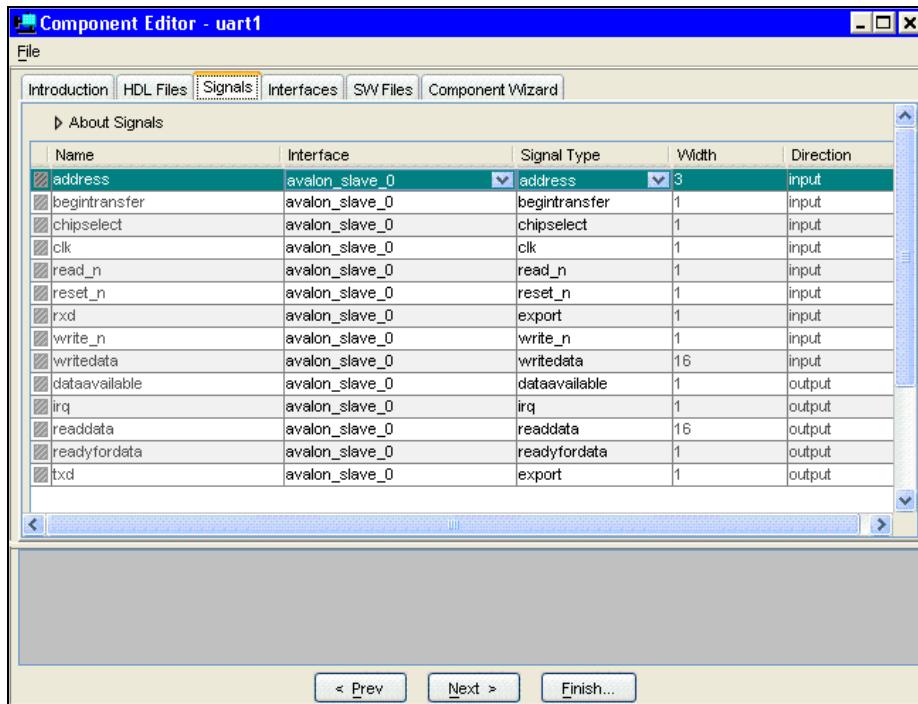
You use the **Signals** tab to specify the purpose of each signal in the top level component module. If you specified files on the **HDL Files** tab, then the signals on the top-level module appear on the **Signals** tab. If the component is an interface to external logic only, then you must manually add the signals that comprise the interface to the external logic.

Each signal must be assigned a signal type. The default 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** column. In addition to Avalon port interfaces that you create, every component has a *global interface* for common signals such as `clk` and `reset`, and for any exported signals.

Figure 5–2 shows an example of the **Signals** tab.

Figure 5–2. Signals Tab



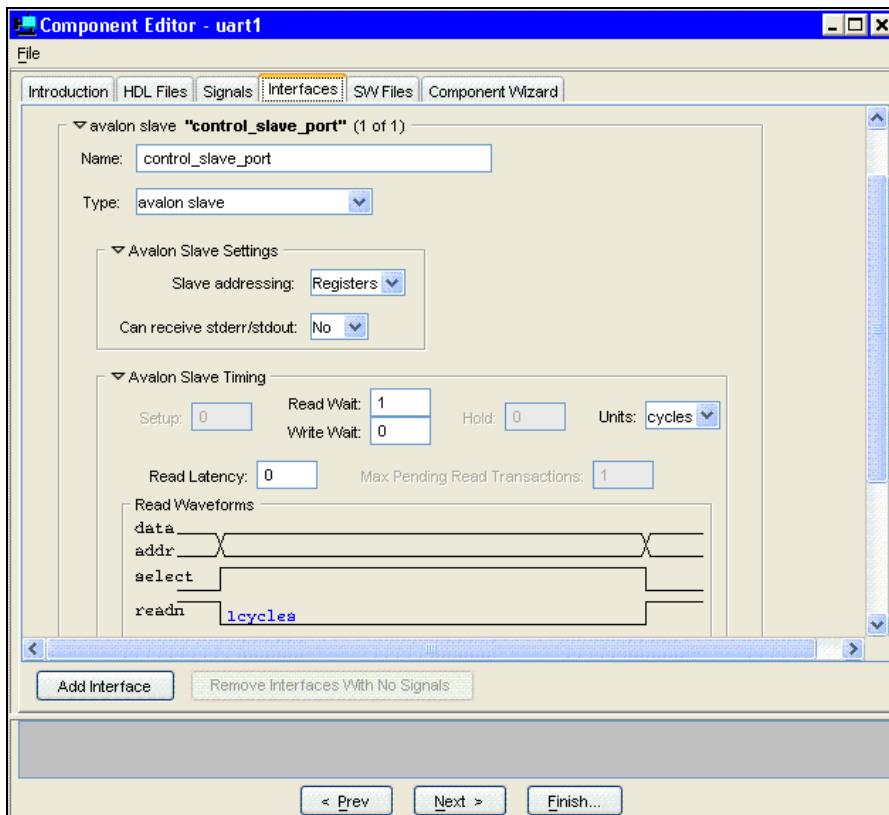
Interfaces Tab

The **Interfaces** tab lets you configure the interfaces on your component, and specify a name for each interface. The interface name identifies the interface, and appears in the SOPC Builder GUI 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 lets you configure the type and properties of each interface. For example, an Avalon slave interface has timing parameters which you must set appropriately.

Figure 5–3 shows an example of the **Interfaces** tab.

Figure 5–3. Interfaces Tab



SW Files Tab

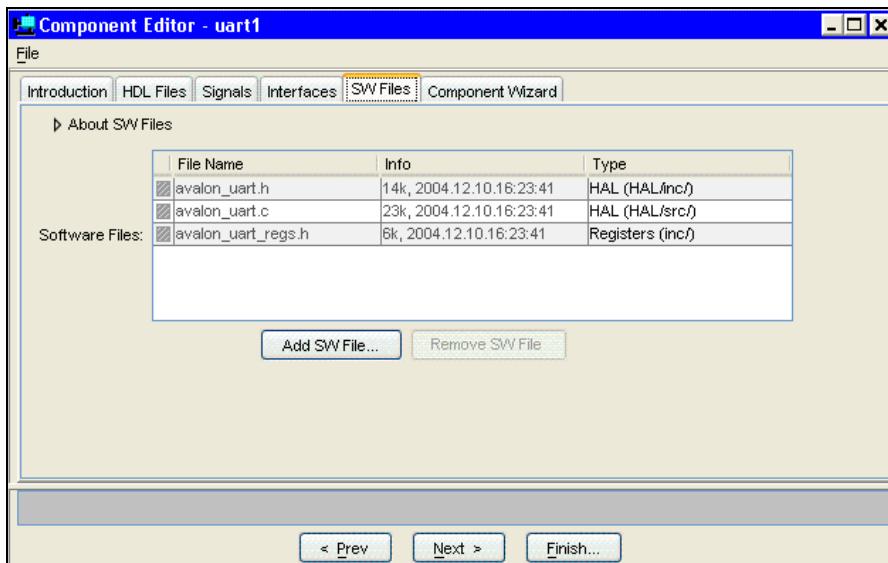
The software files tab (**SW Files**) lets you add existing driver software for your new component. When you save the component later, the component editor copies the software files to software subdirectories inside the component directory. The component editor uses the software directory structure specified by the hardware abstraction layer (HAL) for the Nios II processor.



For information on writing component driver software for the Nios® II processor, including the directory structure, see the *Nios II Software Developer's Handbook*.

Figure 5–4 shows an example of the **SW Files** tab.

Figure 5–4. Software Files (SW Files) Tab

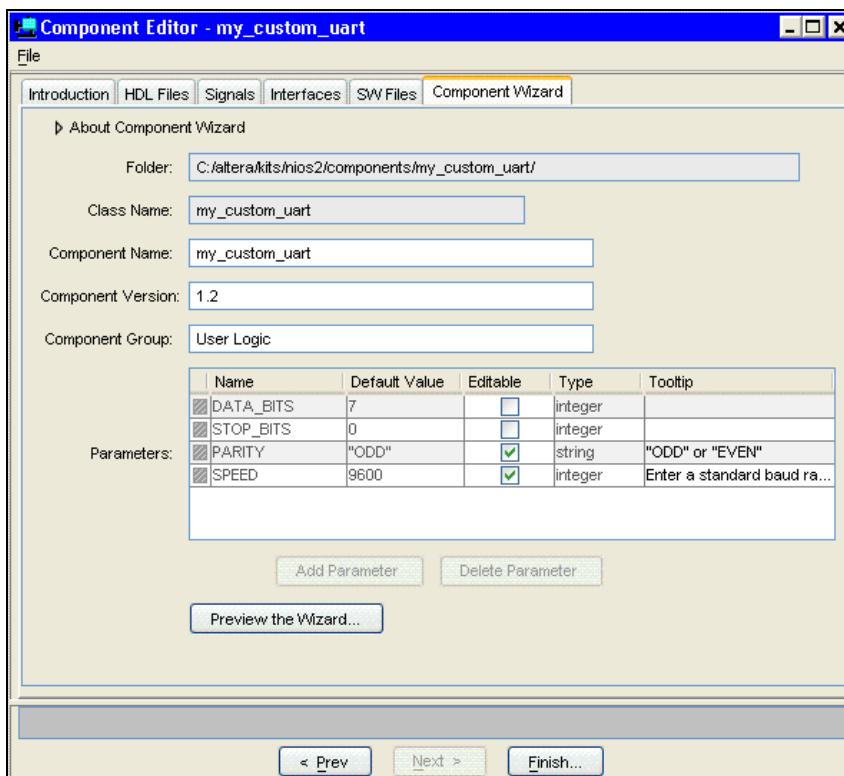


Component Wizard Tab

The **Component Wizard** tab provides settings that affect the presentation of your new component to the user.

Figure 5–5 shows an example of the **Component Wizard** tab.

Figure 5–5. Component Wizard Tab



Identifying Information

You can specify information that identifies the component, such as the component name, version, and group. The component name is the name that appears in the list of available components in the SOPC Builder GUI. The name can include spaces or other special characters.

Based on the component name you specify, the component editor creates a valid class name. The class name determines the component directory name.

You can also specify a component version and a component group. The group setting determines in which group SOPC Builder displays your component in the list of available components. If you enter a previously unused group name, SOPC Builder will display a new group by that name.

Parameters

The **Parameters** table lets you specify the user-configurable parameters for the component.

If the top-level module of the component HDL declares any parameters (*parameters* for Verilog, or *generics* for VHDL) then those parameters appear in the **Parameters** table. These parameters are presented to users when they create or edit an instance of your component.

 To provide parameterization for a lower-level submodule, the top-level module must declare the parameter, and pass it down to the appropriate submodule.

Using the **Parameters** table, you can specify whether or not each parameter is user-editable or not. You can also specify a tooltip that is displayed when a user mouses over the parameter name, to help explain its use.

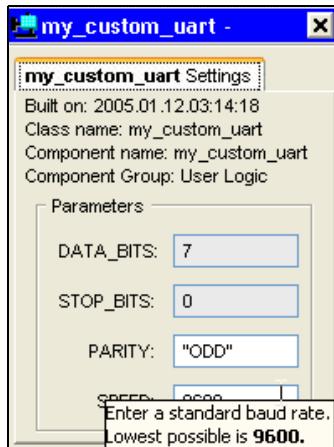
 Tooltips can use basic HTML tags, such as `` and `<p>` to format tooltip text. See [Figure 5–6](#).

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 $N\text{-}1..0$.
- When a VHDL component is used in a Verilog 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 look to an end user. [Figure 5–6](#) shows an example of a component wizard preview.

Figure 5–6. Example Component Wizard Preview



Saving a Component

You can save the component by clicking **Finish** on any of the tabs, or by choosing **Save** (File menu). The component editor saves the component files to the current Quartus® II project directory, in a subdirectory named the same as the component class name specified on the **Component Wizard** tab.

When your component design is final, you can move the component files to a different location.

Re-Editing a Component

After you save a component and exit the component editor, you can re-edit a component at any time from the SOPC Builder GUI. To edit a component, right-click it in the list of available components, and choose **Edit Component**. You cannot edit components that were not created by the component editor, such as Altera®-provided components.

If you alter the source HDL, you need to use the component editor to incorporate the HDL changes into the SOPC Builder component. If you alter the signals on the top-level module, you need to reassign interface signals. If you change the name of a component while editing it, the component editor will save the component files to a new directory.



Existing SOPC Builder systems that use a previous version of the component require further action to reflect the component changes.

To update existing SOPC Builder systems with the new component changes, perform the following steps:

1. Delete any existing instances of the component in the SOPC Builder system, and add the component again.
2. Regenerate the system.



Section II. Building Systems with SOPC Builder

Section II of this volume 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 which 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 6, Developing Components for SOPC Builder](#)
- [Chapter 7, Building Systems with Multiple Clock Domains](#)

Revision History

The following table shows the revision history for Chapters 6–7. These version numbers track the document revisions; they have no relationship to the version of SOPC Builder or the Quartus® II software.

Chapter(s)	Date / Version	Changes Made
6	February 2005 v1.0	Initial release.
7	February 2005 v1.0	Initial release.

Introduction

This chapter describes the design flow to develop a custom SOPC Builder component. This chapter provides tutorial steps that guide you through the process of creating a custom component, integrating it into a system, and downloading it to hardware.

This chapter is divided into the following sections:

- *Component Development Flow* (see page [6–3](#)).
- *Design Example: Pulse-Width Modulator (PWM) Slave* (see page [6–8](#)).
This design example demonstrates developing a component with a single Avalon™ slave interface. In this section, you will start with a ready-made HDL design, package it into a SOPC Builder component, and then instantiate it in a system. If you have a development board, you can download the design to hardware and see the PWM work.
- *Sharing Components* (see page [6–28](#)). This section shows you how to relocate component files to use them in other systems, or share them with other designers.

SOPC Builder Components and the Component Editor

SOPC Builder provides a component editor that lets you create and edit your own SOPC Builder components. By following the procedures described in this document, you will learn to use the component editor and turn any custom logic module into an SOPC Builder component.

Once your custom logic is packaged as component, you can instantiate it in an SOPC Builder system in the same manner as commercially available SOPC Builder Ready components. You can share your component with other designers to encourage design reuse.

Typically, a component is comprised of the following:

- Hardware files: HDL modules that describe the component hardware
- Software files: A C-language header file that defines the component register map, and driver software that allows programs to control the component
- Component description file (`class.ptf`): This file defines the structure of the component, and provides SOPC Builder the information it needs to integrate the component into a system. The component

editor generates this file automatically based on the hardware & software files you provide, and the parameters you specify in the component editor GUI.

After you create the hardware and software files that describe the component, you use the component editor to package those file into an SOPC Builder component. You can also use the component editor later to re-edit the component, if you ever update the hardware or software files.

Assumptions About the Reader

This chapter assumes that you are familiar with the following:

- Building systems with SOPC Builder. For details, see the *Introduction to SOPC Builder* and *Tour of the SOPC Builder User Interface* chapters in Volume 4 of the *Quartus II Handbook*.
- SOPC Builder components. For details, see the *SOPC Builder Components* chapter in Volume 4 of the *Quartus II Handbook*.
- Basic concepts of the Avalon interface. You do not need extensive knowledge of the Avalon interface, such as transfer types or signal timing, to use the design example(s) provided with this chapter. However, to create your own custom components, you need a fuller understanding of the Avalon interface. For details, see the *Avalon Interface Specification*.

Hardware and Software Requirements

To use the design example(s) in this chapter, you must have the following:

- Design files for the example design – A hyperlink to the design files appears next to this chapter on the SOPC Builder literature page. Visit www.altera.com/sopcbuilder.
- Quartus® II Software version 4.2 or higher – Both Quartus II Web Edition and the fully licensed version will work with the example design.
- Nios® II development kit 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® II Edition
 - Stratix Edition
 - Stratix Professional Edition
 - Cyclone™ Edition

If you do not have a development board, you can follow the hardware development steps, but you will not be able to download the complete system to a working board.



You can download the Quartus II Web Edition software and the Nios II Development Kit, Evaluation Edition for free from the Altera Download Center at www.altera.com.



Before you begin, you must install the Quartus® II software and Nios II development tools.

Component Development Flow

This section provides an overview of the development process for SOPC Builder components, covering both the hardware and software aspects. This section focuses on the design flow for components with a single Avalon slave interface. However, these steps are easily extrapolated to components with a master port, or multiple master and slave ports.

Typical Design Steps

A typical development sequence for a slave component includes the following steps, not necessarily in this order:

1. Specify the hardware functionality.
2. If a microprocessor will be used to control the component, specify the application program interface (API) to access and control the hardware.
3. Based on the hardware and software requirements, define an Avalon interface that provides:
 - a. Appropriate control mechanisms
 - b. Adequate throughput performance
4. Write HDL that describes the hardware in either Verilog or VHDL.
5. Test the component hardware alone to verify correct operation.
6. Write a C header file that defines the hardware-level register map for software.
7. Use the component editor to package the initial hardware and software files into a component.

8. Instantiate the component into a simple SOPC Builder system module.
9. Test register-level accesses to the component using a microprocessor, such as the Nios II processor. You can perform verification in hardware, or on an HDL simulator such as ModelSim.
10. If a microprocessor will be used to control the component, write driver software.
11. Iteratively improve the component design, based on in-system behavior of the component:
 - a. Make hardware improvements and adjustments.
 - b. Make software improvements and adjustments.
 - c. Incorporate hardware and software changes into the component using the component editor.
12. Build a complete SOPC Builder system incorporating one or more instances of the component.
13. Perform system-level verification. Make further iterative improvements, if necessary.
14. Finalize the component and distribute it for design reuse.

The design process for a master component is similar, except for software development aspects.

Hardware Design

As with any logic design process, the development of SOPC Builder component hardware begins after the specification phase. Coding the HDL is an iterative process, as you write and verify the HDL logic against the specification.

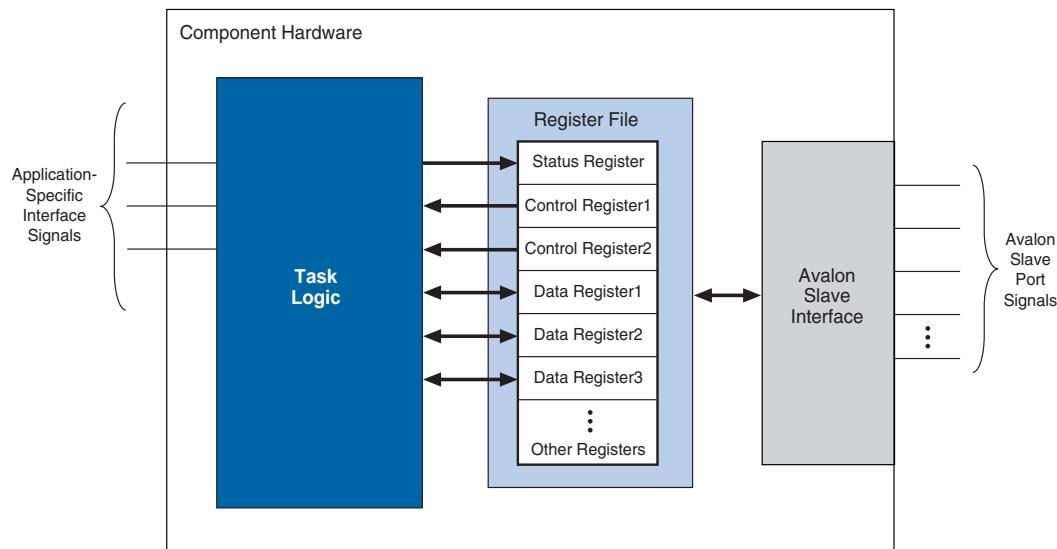
The architecture of a typical component consists of the following functional blocks:

- *Task Logic* - The task logic implements the component's fundamental function. The task logic is design dependent.

- *Register File* - The register file provides a path for communicating signals from inside the task logic to the outside world, and vice versa. The register file maps internal nodes to addressable offsets that can be read or written by the Avalon interface.
- *Avalon Interface* - The Avalon interface provides a standard Avalon front-end to the register file. The interface uses any Avalon signal types necessary to access the register file and support the transfer types required by the task logic. The following factors affect the Avalon interface:
 - How wide is the data to be transferred?
 - What is the throughput requirement for the data transfers?
 - Is this interface primarily for control or for data? That is, do transfers tend to be sporadic, or come in continuous bursts?
 - Is the hardware relatively fast or slow compared to other components that will be in a system?

Figure 6–1 shows a block diagram of a typical component with one Avalon slave port.

Figure 6–1. Typical Component with One Avalon Slave Port



Software Design

If your intent is for 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

slave port that is accessible to a processor. The component editor lets you package a C header file with the component to define the software view of the hardware.

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 an excerpt from the register map for an Altera-provided UART component for the Nios II processor.

Example: Register Map for a Component

```
#include <io.h>
#define IOADDR_ALTERA_AVALON_TIMER_STATUS(base)      __IO_CALC_ADDRESS_NATIVE(base, 0)
#define IORD_ALTERA_AVALON_TIMER_STATUS(base)         IORD(base, 0)
#define IOWR_ALTERA_AVALON_TIMER_STATUS(base, data)    IOWR(base, 0, data)

#define ALTERA_AVALON_TIMER_STATUS_TO_MSK            (0x1)
#define ALTERA_AVALON_TIMER_STATUS_TO_OFST          (0)
#define ALTERA_AVALON_TIMER_STATUS_RUN_MSK          (0x2)
#define ALTERA_AVALON_TIMER_STATUS_RUN_OFST         (1)

#define IOADDR_ALTERA_AVALON_TIMER_CONTROL(base)     __IO_CALC_ADDRESS_NATIVE(base, 1)
#define IORD_ALTERA_AVALON_TIMER_CONTROL(base)        IORD(base, 1)
#define IOWR_ALTERA_AVALON_TIMER_CONTROL(base, data)  IOWR(base, 1, data)

#define ALTERA_AVALON_TIMER_CONTROL_ITO_MSK          (0x1)
#define ALTERA_AVALON_TIMER_CONTROL_ITO_OFST         (0)
#define ALTERA_AVALON_TIMER_CONTROL_CONT_MSK         (0x2)
#define ALTERA_AVALON_TIMER_CONTROL_CONT_OFST        (1)
#define ALTERA_AVALON_TIMER_CONTROL_START_MSK        (0x4)
#define ALTERA_AVALON_TIMER_CONTROL_START_OFST       (2)
#define ALTERA_AVALON_TIMER_CONTROL_STOP_MSK         (0x8)
#define ALTERA_AVALON_TIMER_CONTROL_STOP_OFST        (3)
```

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.

Driver software is dependent on the target processor. The component editor lets you easily package software drivers for the hardware abstraction layer (HAL) used by the Nios II processor development tools. To provide drivers for other processors, you must accommodate the needs of the development tools for the target processor.



For details on writing drivers for the Nios II HAL, see the *Nios II Software Developer's Handbook*. It is instructive to look at the software files provided for other ready-made components. The Nios II development kit provides many components you can use as reference. See <Nios II kit path>/components/.

Verifying the Component

You can verify the component in incremental stages, as you complete more and more of the design. Typically, you first verify the hardware logic as a unit (which might comprise 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 behavioral or register transfer level (RTL) simulation tools. Similarly, you can verify all component logic, including the register file and the Avalon interface(s), using your preferred verification tools.

After you package the HDL files into a component using the component editor, the Nios II development kit 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. The results can be verified either on the ModelSim simulator or on hardware, such as a Nios development board.



See AN351: *Simulating Nios II Embedded Processor Designs* for more information.

System-Level Verification

After you package the HDL files into a component using 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 RTL simulators such as ModelSim. While SOPC Builder produces a testbench for system-level verification, the capability of the simulation environment is largely dependent on the components included in the system.



During the verification phase, including a Nios II processor in the system can be useful to get the benefits of the Nios II simulation environment. Even if your component has no relationship to the Nios II processor, the auto-generated ModelSim simulation environment provides an easy-to-use base that you can build upon to verify other logic in the system.

Design Example: Pulse-Width Modulator Slave

This section uses a pulse-width modulator (PWM) design example to demonstrate the steps to create a component and instantiate it in a system. This component has a single Avalon slave port.

In this section, you will perform the following steps:

1. Install the design files.
2. Review the example design specifications.
3. Package the design files into an SOPC Builder component.
4. Instantiate the component in hardware.
5. Compile the hardware design in the Quartus II software, and download the design to a target board.
6. Exercise the hardware using Nios II software.

Install the Design Files

Before you proceed, you must install the Nios II development tools and download the PWM example design from the Altera web site. The hardware design used in this chapter is based on the **standard** hardware example design included with the Nios II development kit.



Do not use spaces in any directory path names when installing the design files. If the path contains spaces, SOPC Builder might not be able to access the files.

Perform the following steps to setup the design environment:

1. Unzip the contents of the PWM zip file to a directory on your computer. This document will refer to this directory as the <PWM design files> directory.
2. On your host computer file system, locate the following directory:

<Nios II kit path>/examples/<verilog or vhdl>/<board version>/standard

Each development board has a VHDL and Verilog version of the design. You can use either one. **Table 6–1** shows the names of the directories for the available Nios development boards.

Table 6–1. Design File Directories	
Nios Development Board	Tutorial Directory
Stratix II Edition	niosll_stratixll_2s60_es
Stratix Edition	niosll_stratix_1s10 or niosll_stratix_1s10_es
Stratix Professional Edition	niosll_stratix_1s40
Cyclone Edition	niosll_cyclone_1c20

For demonstration purposes, the figures in this chapter show the case of the Verilog design on the Nios Development Board, Cyclone Edition.

3. Copy the **standard** directory to a new location. By copying the design files, you avoid corrupting the original design. This document will refer to the newly-created directory as the *<Quartus II project>* directory.

Review the Example Design Specifications

This section discusses the design specifications for the provided PWM example design, giving details on each of the following topics:

- PWM Design Files
- Functional Specification
- PWM Task Logic
- Register File
- Avalon Interface
- Software API

In a typical design flow, it is the designer's responsibility to specify the behavior of the component.

PWM Design Files

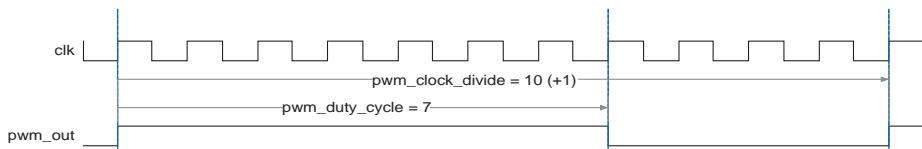
Table 6–2 lists the contents provided in the <PWM design files> directory.

Table 6–2. PWM Design Files Directory	
File Name	Description
/pwm_hw	Contains HDL files describing the component hardware.
pwm_task_logic.v	Contains the core of the PWM functionality.
pwm_register_file.v	Contains logic for reading and writing PWM registers.
pwm_avalon_interface.v	Instantiates task logic and register file, and provides an Avalon slave interface. This file contains the top-level module.
/pwm_sw	Contains C files describing the software interface to the component.
/inc	Contains header files defining low-level hardware interface.
avalon_slave_pwm_regs.h	Defines macros to access registers in the PWM component.
/HAL	Contains HAL driver files for the Nios II processor.
/inc	Contains HAL driver include files.
altera_avalon_pwm_routines.h	Declares function prototypes for accessing the PWM.
/src	Contains HAL driver source code files.
altera_avalon_pwm_routines.c	Defines functions for accessing the PWM.
/test_software	Contains an example program to test the component hardware & software.
hello_altera_avalon_pwm.c	main() initializes the PWM hardware, and uses the PWM to blink an LED.

Functional Specification

A PWM component outputs a square wave with modulated duty cycle. A basic pulse-width waveform is shown in Figure 6–2.

Figure 6–2. Basic Pulse-Width Modulation Waveform



The PWM component is specified and created as follows:

- The task logic operates synchronously to a single clock.
- The task logic uses 32-bit counters to provide a suitable range of PWM periods and duty cycles.
- A host processor is responsible for setting the PWM period value and duty-cycle value. This requirement implies the need for a read/write interface to control logic.
- Register elements are defined to hold the PWM period value and duty-cycle value.
- The host processor can halt the PWM output by using an enable control bit.

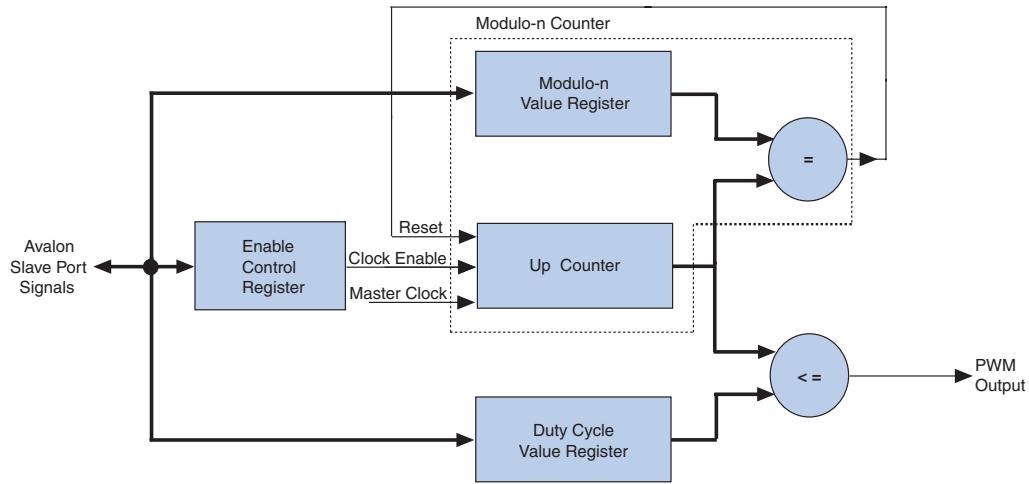
PWM Task Logic

The PWM task logic has the following characteristics:

- The PWM task logic consists of an input clock (`c1k`), an output signal (`pwm_out`), an enable bit, a 32-bit modulo-n counter, and a 32-bit comparator circuit.
- `c1k` drives the 32-bit modulo-n counter to establish the period of the `pwm_out` signal.
- The comparator compares the current value of the modulo-n counter to the duty-cycle value and determines the output of `pwm_out`.
- When the current count value is less than or equal to the duty-cycle value, `pwm_out` drives logic value 0; otherwise, it drives logic value 1.

The task-logic structure is shown in [Figure 6–3](#).

Figure 6–3. PWM Task Logic Structure



Register File

The register file provides access to the enable bit, the modulo-n value and the duty cycle value, shown in [Figure 6–3](#). The design maps each register to a unique offset in the Avalon slave port address space.

Each register has read and write access, which means that software can read back values previously written into the registers. This is an arbitrary design choice that provides software convenience at the expense of hardware resources. You could equally design the registers to be write-only, which would conserve on-chip logic resources, but make it impossible for software to read back the register values.

The register file and offset mapping is shown in [Table 6–3](#). To support three registers, two bits of address encoding are necessary. This gives rise to the fourth register which is reserved.

<i>Table 6–3. Register File & Address Mapping</i>			
Register Name	Offset	Access	Description
clock_divide	00	Read / Write	The number of clock cycles counted during one cycle of the PWM output.
duty_cycle	01	Read / Write	The number of clock cycles in which the PWM output will be low.
enable	10	Read / Write	Enables/disables the PWM output. Setting bit 0 to 1 enables the PWM.
Reserved	11	-	

To read or write the registers requires only one clock cycle, which affects the wait-states for the Avalon interface.

Avalon Interface

The Avalon interface for the PWM component requires a single slave port using a small set of Avalon signals to handle simple read and write transfers to the registers. The component's Avalon slave port has the following characteristics:

- It is synchronous to the Avalon slave port clock.
- It is readable and writeable.
- It has zero wait states for reading and writing, because the registers are able to respond to transfers within one clock cycle.
- It has no setup or hold restrictions for reading and writing.
- Read latency is not required, because all transfers can complete in one clock cycle. Read latency would not improve performance.
- It uses native address alignment, because the slave port is connected to registers rather than a memory device.

Table 6–4 lists the Avalon signals types required to implement these transfer properties. The table also lists the names of each signal as defined in the HDL design file.

Table 6–4. PWM Signal Names & Avalon Signal Types				
Signal Name in HDL	Avalon Signal Type	Bit-Width	Direction	Notes
clk	clk	1	input	Clock that synchronizes data transfers and task logic
resetn	reset_n	1	input	Reset signal; active low.
avalon_chip_select	chipselect	1	input	Chip-select signal
address	address	2	input	2-bit address; only three encodings are used.
write	write	1	input	Write enable signal
write_data	writedata	32	input	32-bit write-data value
read	read	1	input	Read enable signal
read_data	readdata	32	output	32-bit read-data value



For details on the behavior of Avalon signals and Avalon transfers, see the *Avalon Interface Specification*.

Software API

The PWM example design provides both a header file that defines the register map, and driver software for the Nios II processor. See [Table 6–2 on page 6–10](#) for a description of the individual files. The driver functions are listed in [Table 6–5](#).

Table 6–5. PWM Driver Functions (1)	
Function	Prototype Description
altera_avalon_pwm_init();	Initializes the PWM hardware
altera_avalon_pwm_enable();	Activates the PWM output
altera_avalon_pwm_disable();	Deactivates the PWM output
altera_avalon_pwm_change_duty_cycle();	Deactivates the PWM output

Note for Table 6–5:

- (1) Each function takes a parameter that specifies the base address of a specific instance of the PWM component.

Package the Design Files into an SOPC Builder Component

In this section, you will use the SOPC Builder component editor to package the design files into an SOPC Builder component. You will perform the following operations:

1. Open the Quartus II project and start the component editor.
2. Configure the settings on each tab of the component editor.
3. Save the Component.

Open the Quartus II Project & Start the Component Editor

To open SOPC Builder from the Quartus II software, you must have a Quartus II project open. Perform the following steps:

1. Start the Quartus II software.
2. Open the project **standard.qpf** in the *<Quartus II project>* directory.
3. Choose **SOPC Builder** (Tools menu). The SOPC Builder GUI appears, displaying a ready-made example design containing a Nios II processor and several components in the table of active components.
4. Choose **New Component** (File menu). The component editor GUI appears, displaying the **Introduction** tab.

HDL Files Tab

In this section you will associate the HDL files with the component using the **HDL Files** tab. Perform the following steps:

1. Click the **HDL Files** tab.
-  Each tab in the component editor GUI provides on-screen information that describes how to use each tab. Click the triangle at the top-left of each tab to view these instructions.
2. Click **Add HDL File**.
 3. Browse to the *<PWM design files>/pwm_hw* directory. There are three Verilog HDL (.v) files in this directory.
 4. Select all three HDL files in this directory and click **Open**. Use the control key to select multiple files.

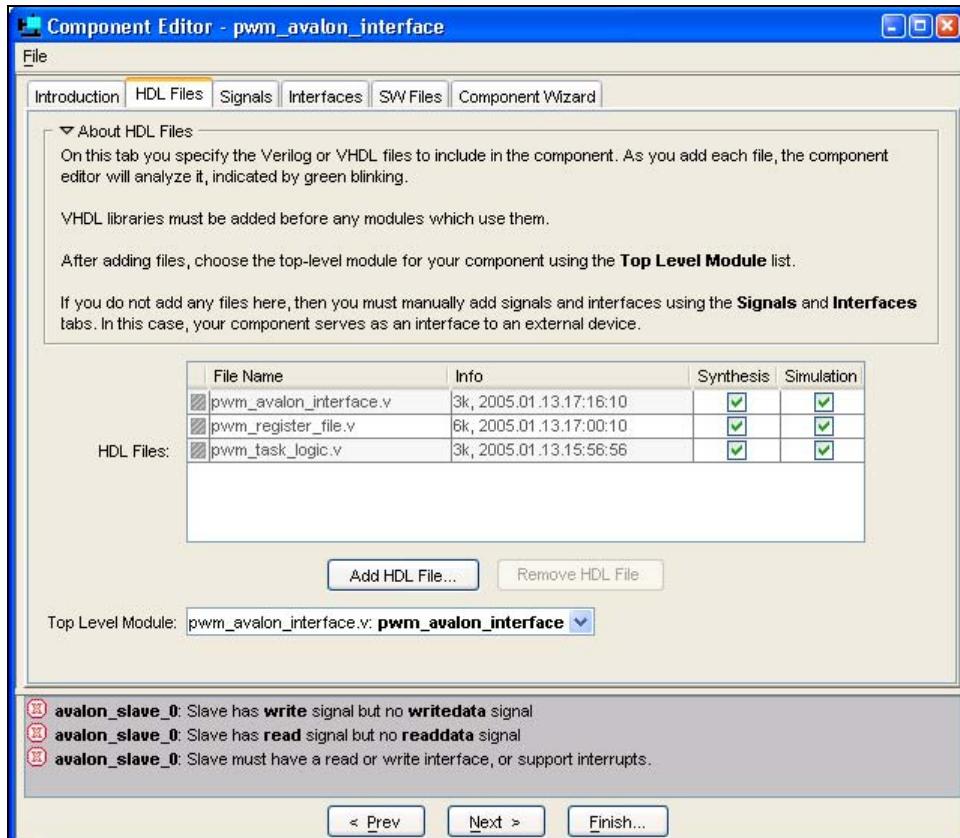
You will return to the **HDL Files** tab. The component editor immediately analyzes each file to read I/O signal and parameter information from the file.

5. Ensure that both the **Simulation** and **Synthesis** boxes are turned on for all files. This indicates that each file is appropriate for both simulation and synthesis design flows.
6. Select **pwm_avalon_interfave.v: pwm_avalon_interface** in the **Top Level Module** list to specify the top-level module.

At this point, the component editor GUI displays error messages. Ignore these messages for now, because you will fix them in later steps.

Figure 6–4 shows the state of the **HDL Files** tab.

Figure 6–4. HDL Files Tab



Signals Tab

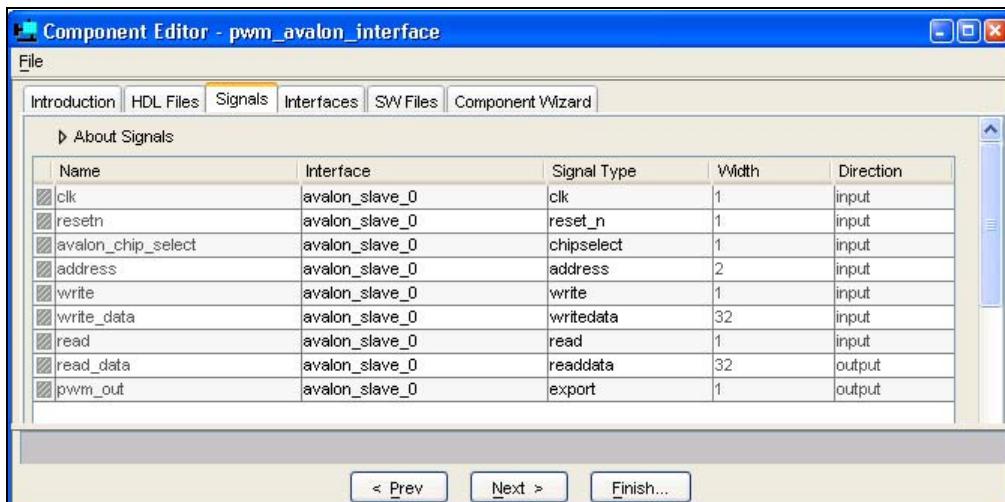
For every I/O signal present on the top-level HDL module, you must map the signal name to a valid Avalon signal type using the **Signals** tab. The component editor automatically fills in signal details that it finds in the top-level HDL source file. If a signal is named the same as a recognized Avalon signal type (such as `write` or `address`), then the component editor automatically assigns the signal's type. If the component editor cannot determine the signal type, it assigns it to type `export`.

Perform the following steps to define the component I/O signals:

1. Click the **Signals** tab. All of the I/O signals in the top level HDL module `pwm_avalon_interface` appear automatically.
2. Assign the **Signal Type** settings for all signals, as shown in [Figure 6–5](#). To change a value, click the **Signal Type** cell to display a drop-down list, and select a new signal type from the list.

After you correctly assign each signal name to a signal type, the error messages should disappear.

Figure 6–5. Assigning Signal Names to Signal Types



You assign type `export` to the signal `pwm_out`, because it is not an Avalon signal. It is intended to be an output of the SOPC Builder system.

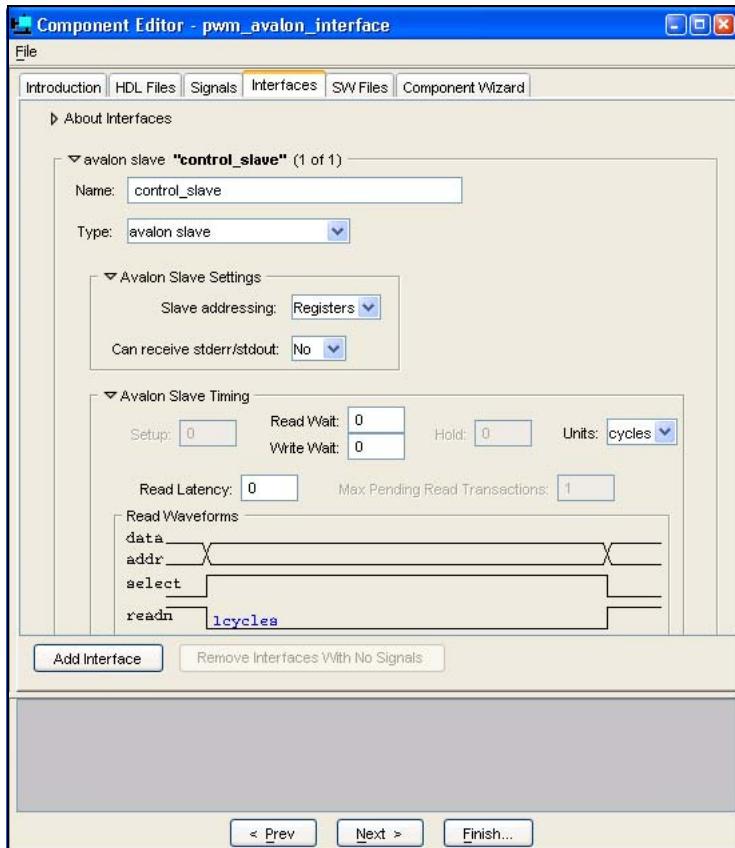
Interfaces Tab

The **Interfaces** tab lets you configure the properties of all Avalon interfaces on the component. In this case there is only one Avalon interface, as specified in the section “[Avalon Interface](#)” on page 6–13. Perform the following steps to configure the Avalon slave port:

1. Click the **Interfaces** tab. The component editor displays a default Avalon slave port that it created automatically, based on the top-level I/O signals in the component design.
2. Type `control_slave` in the **Name** field to rename the slave port. This name appears in the SOPC Builder GUI when you instantiate the component in SOPC Builder.
3. Change the settings for the `control_slave` interface as listed in [Table 6–6](#) below. [Figure 6–6](#) on page 6–19 shows the **Interfaces** tab with the correct settings.

Table 6–6. Control Slave Interface Settings

Setting	Value	Description
Slave addressing	Registers	This setting is appropriate for slave ports used to access address-mapped registers
Read Wait	0	This setting means that the slave port responds to read requests in a single clock cycle (i.e., it does not need read waitstates.)
Write Wait	0	This setting means that the slave port captures write requests in a single clock cycle (i.e., it does not need write waitstates.)

Figure 6–6. Configuring the Interface Properties

Software Files (SW Files) Tab

The **SW Files** tab lets you associate software files with the component, and specify their usage. This component example design provides both a header file that defines the registers and driver software for the Nios II processor. For a description of each file, see [Table 6–2 on page 6–10](#).

Perform the following steps to import the software files into the component:

1. Click the **SW Files** tab.
2. Click **Add SW File**. The Open dialog appears.

3. Browse to the directory <PWM design files>/pwm_sw/inc.
4. Select the file **altera_avalon_pwm_regs.h** and click **Open**.
5. Click the **Type** cell for **altera_avalon_pwm_regs.h** to change the file type. A drop-down list appears.
6. Select **Registers (inc/)**.
7. Repeat steps 2 to 6 to add the file <PWM design files>/pwm_sw/HAL/inc/altera_avalon_pwm_routines.h and set its type to **HAL (HAL/inc/)**.
8. Repeat steps 2 to 6 to add the file <PWM design files>/pwm_sw/HAL/src/altera_avalon_pwm_routines.c and set its type to **HAL (HAL/src/)**.

Figure 6–7 shows the SW Files tab with the correct settings.

Figure 6–7. Software Files (SW Files) Tab

File Name	Info	Type
altera_avalon_pwm_regs.h	3k, 2004.12.17.16:21:10	Registers (inc/)
altera_avalon_pwm_routines.h	2k, 2004.12.17.16:04:28	HAL (HAL/inc/)
altera_avalon_pwm_routines.c	3k, 2004.12.17.16:27:04	HAL (HAL/src/)

Component Wizard Tab

This tab lets you control how SOPC Builder presents the component to a user. Perform the following steps to configure the user presentation of the component:

1. Click the **Component Wizard** tab.
2. For this example, do not change the default settings for **Component Name**, **Component Version**, and **Component Group**.

These settings affect how SOPC Builder identifies the component and displays it in the list of available components. The component editor creates a default name for the component, based on the name of the top-level design module.

3. Under **Parameters**, in the **Tooltip** cell for the parameter **clock_divide_reg_init**, type the following:

```
Initial PWM Period After Reset
```

4. In the **Tooltip** cell for `clock_cycle_reg_init`, type:

Initial Duty Cycle After Reset

5. Click **Preview the Wizard** to preview how the component wizard will appear when instantiated from within SOPC Builder.
6. Close the preview window when you are done.

Save the Component

Perform the following steps to save the component and exit the component editor:

1. Click **Finish**. A dialog appears describing the files that will be created for the component.
2. Click **Yes** to save the files. The component editor saves the files to a subdirectory under `<Quartus II project>`. The component editor closes, and you return to the main SOPC Builder GUI.
3. Locate the new component `pwm_avalon_interface` in the list of available components under the **User Logic** group.

You are ready to instantiate the component into an SOPC Builder system.

Instantiate the Component in Hardware

At this point, the new component is ready to instantiate in an SOPC Builder system. The usage of a component is design dependent, based on the needs of the system. The remaining steps for this design example show one possible way to instantiate and test the component. However, there is an unlimited number of ways this component can be used in a system.

In this section you will add the new PWM component to a system, recompile the hardware design, and configure the FPGA. This section includes the following steps:

1. Add a PWM component to the SOPC Builder system and regenerate the system.
2. Modify the Quartus II design to connect the PWM output to an FPGA pin.
3. Compile the Quartus II design and configure the FPGA with the new hardware image.

Add a PWM Component to the SOPC Builder System

Perform the following steps to setup SOPC Builder's component search path:

1. In the SOPC Builder GUI, choose **SOPC Builder Setup** (File menu).
2. Under **Component/Kit Library Search Path**, enter the path to the <*Quartus II project*> directory. If there are pre-existing paths, use "+" to separate the path names.
3. Click **OK**.



The steps above make the component's software files visible to the Nios II IDE in later steps. These steps are necessary for the Quartus II software v4.2 and the Nios II IDE v1.1. Future releases will eliminate the need for these steps.

Perform the following steps to add a PWM component to the SOPC Builder system:

1. On the SOPC Builder **System Contents** tab, select the new component **pwm_avalon_interface** under the **User Logic** group in the list of available components, and click **Add**. The configuration wizard for the PWM component appears.

If you want to, you can modify the parameters in the configuration GUI. The parameters affect the reset state of the PWM control registers, but have no affect on the outcome of the steps in this chapter.

2. Click **Finish**. You return to the SOPC Builder **System Contents** tab, and the component **pwm_avalon_interface_0** appears in the table of active components.
3. Right-click **pwm_avalon_interface_0** and choose **Rename**.
4. Type **z_pwm_0** for the component name and press **Enter**. (This name is unusual, but it minimizes effort later when you update the Quartus II design in section “[Modify the Quartus II Design to Use the PWM Output](#)” on page [6-23](#).)



You must name the component exactly as directed, or else later steps in this chapter will fail.

5. Click **Generate** to start generating the system.

6. After system generation completes successfully, exit SOPC Builder and return to the Quartus II software.

Modify the Quartus II Design to Use the PWM Output

At this point, you have created an SOPC Builder system that uses the PWM component. Now you must update the Quartus II project to use the PWM output.

The file **standard.bdf** is the top-level Block Design File (BDF) for the Quartus II project. The BDF contains a symbol for the SOPC Builder system module, named **std_<FPGA>**, where **<FPGA>** refers to the FPGA on the target development board.

In the previous steps you added a PWM component which produces an additional output from the system module. Now you need to update the symbol for the system module, and connect the PWM output to an FPGA pin.



To complete this section, you must be familiar with the Quartus II Block Editor.

1. In the Quartus II software, open the file **standard.bdf**.
2. Right-click the symbol **std_<FPGA>** in the BDF and choose **Update Symbol or Block**. The Update Symbol or Block dialog appears.
3. Select **Selected Symbol(s) or Block(s)**.
4. Click **OK** to close the dialog. The symbol **std_<FPGA>** in the BDF is updated, and it now has an additional output port named **pwm_out_from_the_z_pwm_0**.



SOPC Builder creates unique names for all I/O ports on the system module, by combining the signal name in the component design file with the instance name of the component in the system module.

5. Delete the symbol for pins **LEDG[7..0]** which are connected to port **out_port_from_the_led_pio[7..0]** on the system module.

These pins connect to LEDs on the development board. This example design uses one of the LEDs to display the output of the PWM.

6. Create a new output pin named **LEDG[0]**.

7. Connect the new pin LEDG [0] to pwm_out_from_the_z_pwm_0 on std_<FPGA>.

The hardware design is now ready to compile.

Compile the Hardware Design and Download to the Target Board

Perform the following steps to compile the hardware design and download it to the target board.

1. Choose **Save** (File menu) to save changes to the BDF.
2. Choose **Start Compilation** (Processing menu) to start compiling the hardware design. The compilation begins.

If you performed all prior steps correctly, the Quartus II compilation will finish successfully after several minutes, and generate a new FPGA configuration file for the project.



You can only perform the remaining steps in this chapter if you have a development board.

Perform the following steps to download the hardware design to the board:

1. Connect your host computer to the development board using an Altera download cable, such as the USB Blaster, and apply power to the board.
2. Choose **Programmer** (Tools menu) to open the Quartus II Programmer.
3. 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 PWM example design is based on the Nios II processor. You must execute software on the Nios II processor to exercise the PWM hardware. The example design files provide a C test program that pulses an LED by gradually modulating the PWM duty cycle. This test program accesses the hardware both by using the register map declarations directly, and by calling the driver functions.

In this section you will 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.

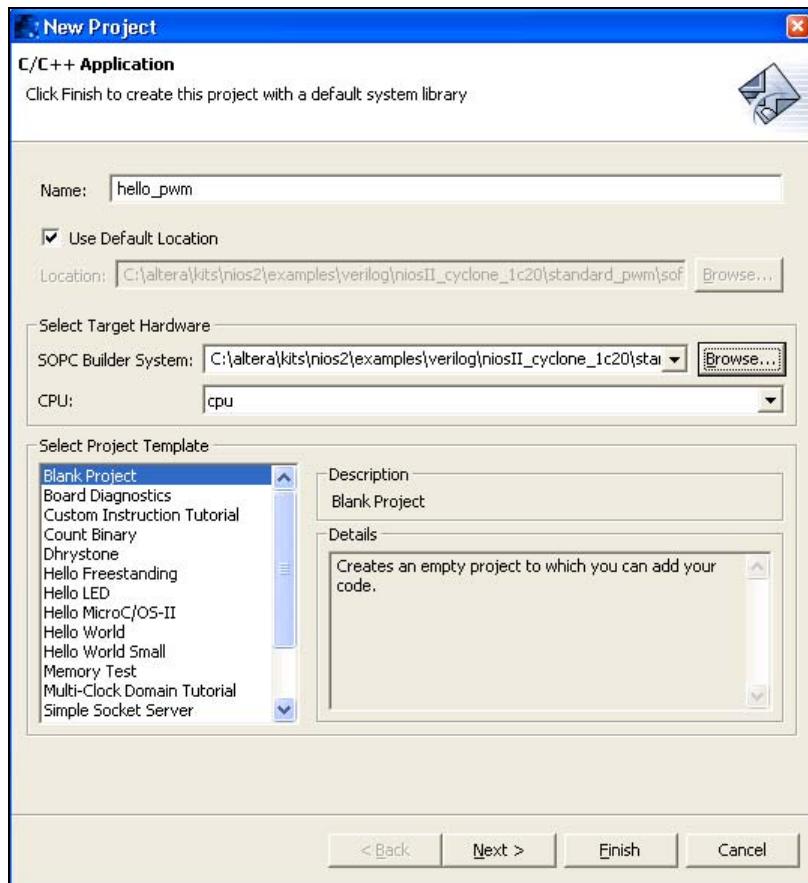
To complete this section, you must have performed all prior steps, and successfully configured the target board with the hardware design.

Start the Nios II IDE & Create a New IDE Project

Perform the following steps to start the Nios II IDE and create a new IDE project:

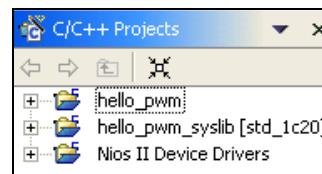
1. Start the Nios II IDE.
2. Choose **New > C/C++ Application** (File menu) to start a new project. The first page of the **New Project** wizard appears.
3. Under **Select Project Template**, select **Blank Project**.
4. In the **Name** field type `hello_pwm`.
5. Ensure that **Use Default Location** is turned on.
6. Click **Browse** under **Select Target Hardware**. The **Select Target Hardware** dialog box appears.
7. Browse to the *<Quartus II project>* directory.
8. Select the file `std_<FPGA>.ptf`.
9. Click **Open** to return to the New Project wizard. The **SOPC Builder System** and the **CPU** fields are now specified, as shown in [Figure 6–8 on page 6–26](#).
10. Click **Finish**.

Figure 6–8. New Project Wizard



After the IDE successfully creates the new project, the C/C++ Projects view will contain two new projects, **hello_pwm** and **hello_pwm_syslib**, in addition to **Nios II Device Drivers**, as shown in Figure 6–9.

Figure 6–9. New Projects in the C/C++ Projects View



Compile the Software Project and Run on the Target Board

In this section you will compile the C test program provided with the PWM design files, and then download it to the target board.

First, perform the following steps to associate the C source file with the new C/C++ project.

1. In your computer's file system, copy the file <PWM design files>/pwm_sw/test_software/hello_altera_avalon_pwm.c to the directory <Quartus II project>/software/hello_pwm/.
2. In the Nios II IDE C/C++ Projects view, right-click **hello_pwm** and choose **Refresh**. This forces 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 **hello_pwm** and choose **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 **hello_pwm** in the C/C++ Projects view.
3. Choose **Run** (Run menu). The Run dialog appears.
4. Under **Configurations** select **Nios II Hardware**, and click **New**. A new run/debug configuration named **hello_pwm Nios II HW configuration** appears.
5. If the **Run** button (in the bottom right of the Run dialog) is deactivated, perform the following steps:
 - a. Click the **Target Connection** tab.
 - b. Click **Refresh** next to the **JTAG cable** list.
 - c. From 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:

- a. The **Console** view in the IDE displays messages similar to the following:

```
Hello from the PWM test program.  
The starting values in the PWM registers are:  
Period = 0  
Duty cycle = 0  
Notice the pulsing LED on the development board.
```

- b. LED0 on the development board repeatedly pulses on and off.

Congratulations! You have finished all steps for the PWM design example.

Sharing Components

When you create a component using the component editor, SOPC Builder automatically saves the component in the current Quartus II project directory. 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.



The directory path name cannot contain spaces. If the path contains spaces, SOPC Builder might not be able to access the files.

2. In SOPC Builder, choose **SOPC Builder Setup** (File menu). The **SOPC Builder Setup** dialog appears, which lets you specify where SOPC Builder searches for component files.
3. Under **Component/Kit Library Search Path**, add the path to the enclosing directory of the component directory. For example, for a component directory **c:\my_component_library\pwm_avalon_interface**, add the path **c:\my_component_library**. If there are pre-existing paths, use "+" to separate the path names.
4. Click **OK**.

QII54008-1.0

Introduction

This chapter guides you through the process of using SOPC Builder to create a system with multiple clock domains. You will start with a ready-made design that uses a single clock domain, and modify the design to use two clocks.

Example Design Overview

The design in this chapter mimics the common scenario of a system with separate control and data paths. Typically, the control path is slow, because the controller itself is relatively slow, and it is used only in short bursts to set up data transfers. On the other hand, the data path is fast so that, after the controller initiates a transfer, data moves as quickly as possible from source to destination.

[Figure 7–1 on page 7–2](#) shows a simplified block diagram of the system structure. In this design, a Nios® II processor acts as the controller operating at 50 MHz. A DMA controller operating at 100 MHz manages the data path, and reads and writes data buffers that also operate at 100 MHz. The figure focuses on the multi-clock nature of the system, and shows the connections between master and slave ports.

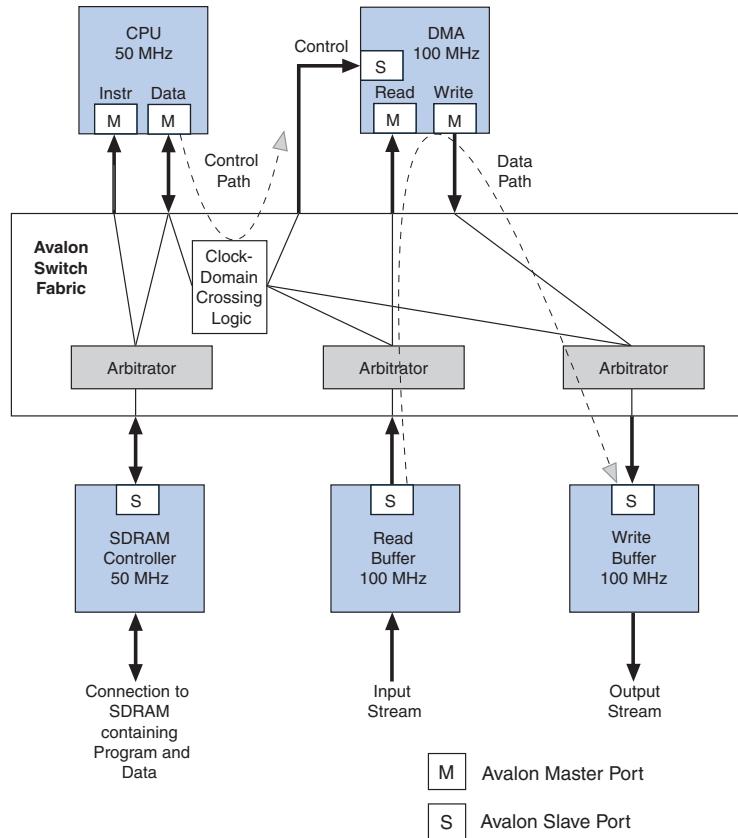
Figure 7–1. Simplified Block Diagram of the Example Design

Figure 7–1 does not show example design features that are not directly related to the issue of multiple clocks, such as a JTAG UART communication peripheral and timer peripheral used by the Nios II processor.

Hardware and Software Requirements

To use the design example(s) in this chapter, you must have the following:

- Design files for the example design – A hyperlink to the design files is located with this chapter on the SOPC Builder literature page. Visit www.altera.com/sopcbuilder.

- Quartus® II Software version 4.2 or higher – Both Quartus II Web Edition and the fully licensed version will work with the example design.
- Nios II development kit 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® II Edition
 - Stratix Edition
 - Stratix Professional Edition
 - Cyclone™ Edition

If you do not have a development board, you can follow the hardware development steps, but you will not be able to download the complete system to a working board.



You can download the Quartus II Web Edition software and the Nios II Development Kit, Evaluation Edition for free from the Altera Download Center at www.altera.com.



Before you begin, you must install the Quartus II software and Nios II development tools.

Creating the Multi-Clock Hardware System

In this section, you will start with an example hardware design provided with the Nios II development kit and create a multi-clock system. You will perform the following steps:

1. Copy the hardware design files to a new directory
2. Modify the design in SOPC Builder to create a multi-clock hardware system
3. Update the Quartus II design to use the new clock domain
4. Compile the hardware design in the Quartus II software, and download the hardware design to a target board

Copy the Hardware Design Files to a New Directory

The hardware design used in this chapter is based on the **standard** hardware example design included with the Nios II development kit. Copy the design files by performing the following steps:

1. In your host computer file system, locate the following directory:

<Nios II kit path>\examples\<verilog or vhdl>\<board version>\standard

Each development board has a VHDL and Verilog version of the design. You can use either one. **Table 7–1** shows the names of the directories for the available Nios development boards.

Table 7–1. Design File Directories	
Nios Development Board	Tutorial Directory
Stratix II Edition	niosll_stratixll_2s60_es
Stratix Edition	niosll_stratix_1s10 or niosll_stratix_1s10_es
Stratix Professional Edition	niosll_stratix_1s40
Cyclone Edition	niosll_cyclone_1c20

For demonstration purposes, the figures in this chapter show the case of the Verilog design on the Nios Development Board, Cyclone Edition.

2. Copy the **standard** directory to a new location. This document will refer to the newly-created directory as *<hardware files directory>*.

Modify the Design in SOPC Builder

This section walks you through the process of implementing a multi-clock system in SOPC Builder. In this section you will do the following:

1. Open the system in SOPC Builder.
2. Add a DMA controller and two 4 Kbyte on-chip memory components.
3. Connect DMA master ports to memory slave ports.
4. Make clock domain assignments.
5. Regenerate the system.

Open the System in SOPC Builder

To open the system in SOPC Builder, perform the following steps:

1. Start the Quartus II software.
2. Choose **Open Project** (File menu).
3. Browse to *<hardware files directory>*.
4. Select **standard.qpf** and click **Open**.
5. Choose **SOPC Builder** (Tools menu) to start SOPC Builder.

The SOPC Builder window appears, displaying the contents of the system, which you will use as a starting point for your design.

Add DMA Controller and Memory Components

Perform the following steps to add the DMA controller and memory components to the system:

1. In the SOPC Builder list of available components, select **DMA** in the **Other** group, and click **Add**. The DMA configuration wizard displays.
2. In the DMA configuration wizard, click **Finish** to accept the default settings. You return to the SOPC Builder **System Contents** tab which displays the new DMA component, named **dma_0**.

Errors are displayed in the SOPC Builder messages window. You can ignore these messages for now, because you will fix the errors in later steps.

3. In the list of available components, select **On-Chip Memory (RAM or ROM)** in the **Memory** group, and click **Add**. The On-Chip Memory configuration wizard appears.
4. In the On-Chip Memory configuration wizard, click **Finish** to accept the default settings. You return to the SOPC Builder **System Contents** tab, which displays the new memory component.
5. Right-click the new memory component and choose **Rename**.
6. Type `read_buffer` to rename the component.
7. Repeat steps 3 and 4 to add another On-Chip Memory component to the system.
8. Right-click the new memory component and choose **Rename**.

- Type `write_buffer` to rename the component.



You must name the DMA and memory components exactly as specified above (`dma_0`, `read_buffer`, and `write_buffer`). If you name the components differently, later steps will fail.

Connect DMA Master Ports to Memory Slave Ports

Now that you have added the DMA controller and the memory components to the system, you must connect their master and slave ports appropriately. Perform the following steps:

- Hover the mouse pointer over the connections panel in the SOPC Builder **System Contents** tab to display the potential master port connections for the DMA. See [Figure 7-2](#).
- Connect the DMA read master port (`dma_0/read_master`) to the `read_buffer` memory.
- Connect the DMA write master port (`dma_0/write_master`) to the `write_buffer` memory.
- Disconnect the Nios II processor instruction master (`cpu/instruction_master`) from both the on-chip memories' slave ports. For this example design, the processor does not use these memories to fetch instructions.
- Verify that the Nios II processor data master (`cpu/data_master`) is connected to the DMA slave port (`dma/control_port_slave`).

[Figure 7-2](#) shows the state of the connections panel after all components have been correctly connected.

Figure 7-2. Correct Connections Between Master & Slave Ports



After the DMA controller is connected properly to the on-chip memories, the error messages disappear from the SOPC Builder messages window.

Make Clock Domain Assignments

In this section you will specify a 100 MHz clock input, and assign the DMA and memory components to the new clock domain. Then you will generate the system. Perform the following steps:

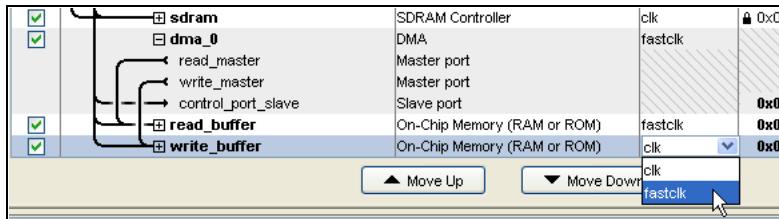
1. On the **System Contents** tab under **Clock MHz**, click the cell labeled **click to add** to enter a new clock entry.
2. Type **fastclk** for the name of the new clock, as shown in Figure 7–3.
3. Type **100** for the speed of **fastclk**, as shown in Figure 7–3.

Figure 7–3. Clock Settings



4. In the **Clock** list for the DMA component, select **fastclk** to assign the 100 MHz clock to the DMA component, as shown in Figure 7–4.
5. Repeat step 4 for the **read_buffer** and **write_buffer** components, as shown in Figure 7–4.

Figure 7–4. Assigning a Different Clock to the DMA & Memory Components



6. Click **Generate** to start generating the system.
7. After system generation completes successfully, exit SOPC Builder and return to the Quartus II software.

Update the Quartus II Design

At this point, you have created an SOPC Builder system that uses multiple clock domains. In this section you will do the following:

1. Update the system module symbol.
2. Update PLL settings to generate a 100 MHz clock.
3. Connect the 100 MHz clock to the system module
4. Compile the Design and download it to the board

To complete this section, you must be familiar with the Quartus II Block Editor.

Update the System Module Symbol

The file **standard.bdf** is the top-level Block Diagram File (BDF) for the Quartus II project. The BDF contains a symbol for the SOPC Builder system module, named **std_<FPGA>**, where **<FPGA>** refers to the FPGA on the target development board.

The SOPC Builder system module requires an additional clock input for the 100 MHz clock domain, and therefore you need to update the symbol for the system module. To remove the old symbol and insert an updated symbol, perform the steps below.

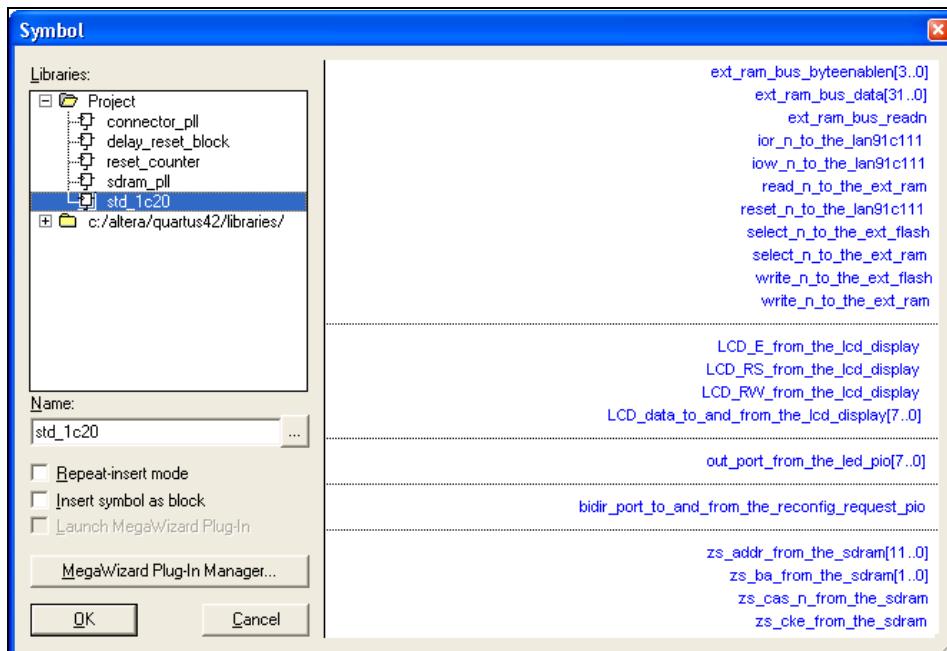


In the following steps you will disconnect and then reconnect all connections to the system module, including connections to FPGA pins. You must make these connections exactly, or else the design will not work in hardware, potentially damaging the development board. If possible, print the BDF as a reference to help you reconnect pins to the system module later.

1. In the BDF, select the symbol **std_<FPGA>** and delete it.
2. Click-and-drag to select all of the pins that previously connected to the right-hand-side of the symbol.
3. Press the right-arrow key ten times to move the pins to the right, creating space for the new symbol.
4. Double click in the space where the symbol used to be to insert a new symbol. The **Symbol** window appears.
5. Expand the **Project** folder under **Libraries**.

6. Select the `std_<FPGA>` symbol, and click OK. See Figure 7–5.

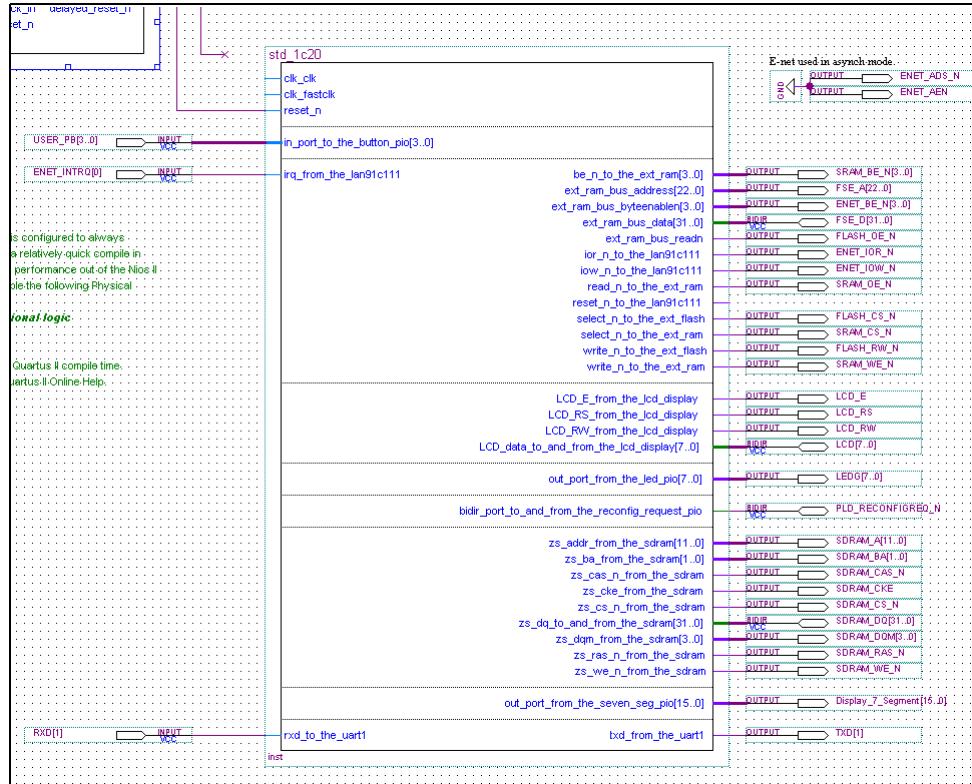
Figure 7–5. Selecting and Inserting the New Symbol



7. Move the mouse to position the symbol between the pin symbols, then click the left mouse button to place the symbol.
8. Look at the symbol, and notice the new clock input, `clk_fastclk`, and the old clock input, which is now named `clk_clk`.
9. Reconnect all previous connections to the `std_<FPGA>` symbol, except for the clock inputs `clk_clk` and `clk_fastclk`. You will connect the clock inputs in later steps.

Figure 7–6 shows the new symbol reconnected to all signals except for the clocks.

Figure 7–6. Updated Symbol without Clock Connections



Update PLL Settings to Generate a 100 MHz Clock

Perform the following steps to modify the PLL instance in the BDF to generate a 100 MHz clock:

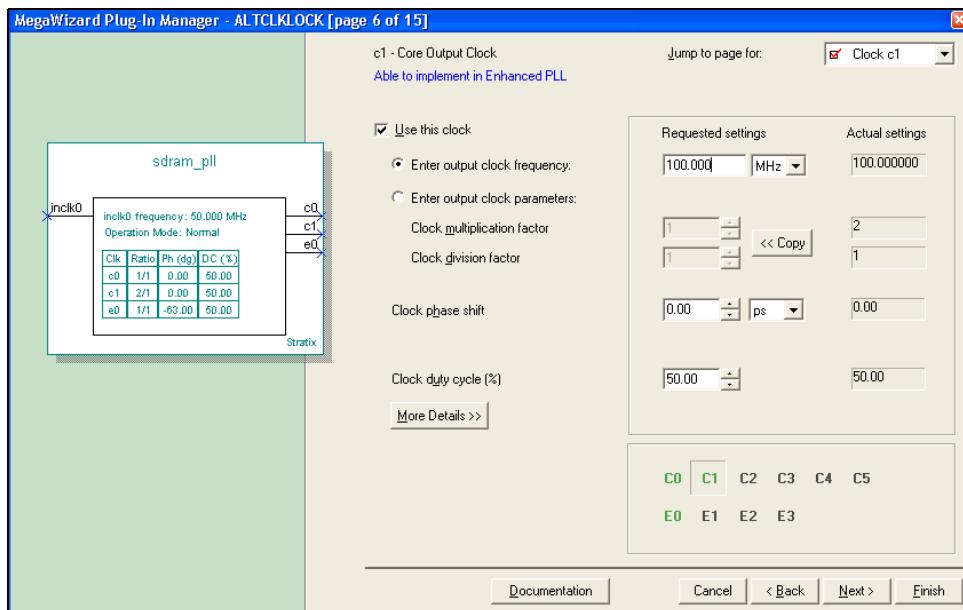
1. Locate the symbol **sdram_pll** in the BDF.
2. Right-click the symbol and select **MegaWizard Plug-in Manager** to configure the PLL settings. The MegaWizard for the ALTCLKLOCK function displays.

At this point, the Quartus II software might display a benign warning: "Delay shifts (time delay elements) are no longer supported in the Stratix PLLs." If you see this message, click **OK**.

3. Click **Next** until you reach page 6 of 15 of the wizard flow, shown in [Figure 7–7](#).
4. Click **Use this clock**.
5. Select **Enter output clock frequency**.
6. Under **Requested settings**, type 100 and select **MHz**.

[Figure 7–7](#) shows the **ALTCLKLOCK** MegaWizard with the correct settings.

Figure 7–7. ALTCLKLOCK Settings to Generate a 100MHz Clock



7. Click **Finish** to jump to the final stage (page 15 of 15) of the wizard flow.
8. Click **Finish** again to return to the Quartus II software.

9. Right-click the **sdram_pll** symbol and choose **Update Symbol or Block**. The Update Symbol or Block dialog appears.
10. Click **OK** to update the selected instance of the symbol.

The PLL is now configured correctly to generate a 100 MHz output clock.



For further information on using the PLLs in Altera devices, see the handbook for the target device family.

Connect the 100 MHz clock to the system module

You now must connect the clock signals to the SOPC Builder system module. The easiest way to accomplish this is to symbolically link the PLL outputs to the system module inputs using conduit aliases. This section will guide you through the process of making one connection, and leave the remaining connections to you as an exercise.

To connect the 100 MHz clock signal from the PLL to the system module, perform the following steps:

1. Move the mouse pointer over node **c1** on **sdram_pll** until the pointer changes to a cross-hairs.
2. Click and drag right to add a conduit (i.e. a connection line) to node **c1**.
3. Click on the conduit line to select it.
4. Type **dmaclk** and press **Enter**.
5. Move the mouse pointer over node **clk_fastclk** on **std_<FPGA>** until the pointer changes to a cross-hairs.
6. Click and drag left to add a conduit to node **clk_fastclk**.
7. Click on the conduit line to select it.
8. Type **dmaclk** and press **Enter**.

The two nodes are now linked symbolically (by the name **dmaclk**) via a conduit alias.

Follow the instructions below to complete the remaining clock connections. Depending on which Nios development board you are targeting, the remaining PLL connections to the PLL(s) are slightly different. This section gives instructions to accommodate all Nios development boards.



Be sure to use the instructions that apply to your board, or else the design will fail in hardware.

For the Nios Development Board, Stratix II Edition, Stratix Professional Edition, and Stratix Edition, connect the PLL according to [Table 7–2](#).

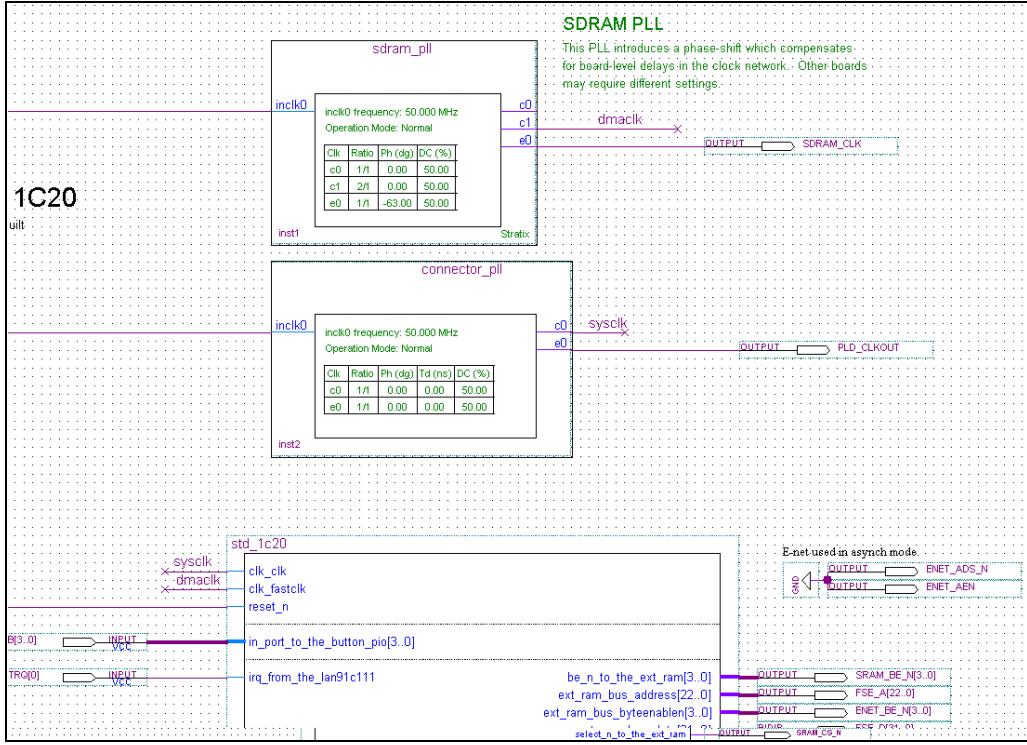
<i>Table 7–2. PLL Connections</i>	
Node on sdram_pll	Connects to
c0	clk_clk on std_<FPGA>
c1	clk_fastclk on std_<FPGA>
e0	PLD_CLKOUT pin

The BDF for the Nios Development Board, Cyclone Edition contains a second PLL symbol named **connector_pll** in addition to **sdram_pll**. Connect both PLLs according to [Table 7–3](#).

<i>Table 7–3. PLL Connections for the Nios Development Board, Cyclone Edition</i>	
Node	Connects to
c0 on sdram_pll	Nothing
c1 on sdram_pll	clk_fastclk on std_<FPGA>
e0 on sdram_pll	SDRAM_CLK pin
c0 on connector_pll	clk_clk on std_<FPGA>
e0 on connector_pll	PLD_CLKOUT pin

Figure 7–8 shows an example of the BDF for the Nios Development Board, Cyclone Edition with all connections completed using conduit aliases.

Figure 7–8. Symbolic Conduit Aliases Connecting PLL(s) to System Module



Compile the Design and Download to the Board

To compile the design and download it to the target board, perform the following steps:

1. Chose **Save** (File menu) to save changes to the BDF.
2. Choose **Start Compilation** (Processing menu) to start compiling the hardware design. The compilation begins.

If you performed all prior steps correctly, the Quartus II compilation will finish successfully after several minutes, and generate a new FPGA configuration file for the project.



You can only perform the remaining steps in this chapter if you have a development board.

To download the hardware design to the board, perform the following steps:

1. Connect your host computer to the development board via an Altera download cable, such as the USB Blaster.
2. Choose **Programmer** (Tools menu) to open the Quartus II Programmer.
3. Use the Programmer window to download the following FPGA configuration file to the board:
<Hardware files directory>\standard.sof.

You have completed all the steps to create a multi-clock hardware design and download it to hardware. This design is based on the Nios II processor, and therefore you will have to run a software program on the processor to exercise the hardware.

Running Software to Exercise the Multi-Clock Hardware

In this section, you will run a program on the Nios II processor to exercise the multi-clock domain hardware. This program sets up and initiates a DMA transfer using the Nios II processor, and measures the time for the DMA transfer to complete.



There is nothing special about this program that makes it specific to multi-clock domain systems. Because the Avalon switch fabric abstracts the details of clock domain crossing, the Nios II processor benefits from the fast performance of the DMA controller without needing to be aware of the system clock domain properties.

You will perform the following steps:

1. Install the example software design files.
2. Create a new Nios II IDE project using the software files.
3. Build and run the program.
4. Analyze the results.

To complete this section, you must have performed all prior steps, and successfully configured the target board with your multi-clock hardware design.

Install the Example Software Design Files

In this section, you will install the example software design files on your computer. You can download the example design files from the Altera web site. Before you proceed installing the files, download the file **multi_clock.zip** associated with the URL for this chapter.

The file **multi_clock.zip** contains the following C-language source files:

- **dma_xfer.c** — Contains `main()`.
- **init.c, init.h** — Contain initialization routines to set up memory buffers, and initialize the timer.
- **settings_check.h** — Provides basic error checking to verify that the hardware contains the necessary DMA and memory components.

The file **multi_clock.zip** contains example software files packaged as a Nios II IDE software template. Perform the following steps to install the files:

1. Extract the contents of **multi_clock.zip** into a new directory called **multi_clock**.
2. Move the **multi_clock** directory under the following directory:
`<Nios II kit path>\examples\software`

The files are packaged as a Nios II IDE template only to minimize the number of steps required for you to run the software. Using a template is not a necessary part of writing software for multi-clock domain systems. Using the template automatically provides the following functionality in the Nios II IDE, which you otherwise would have to perform manually:

1. Imports source files into a new project directory.
2. Sets up the system library settings.
3. Sets up the project settings.

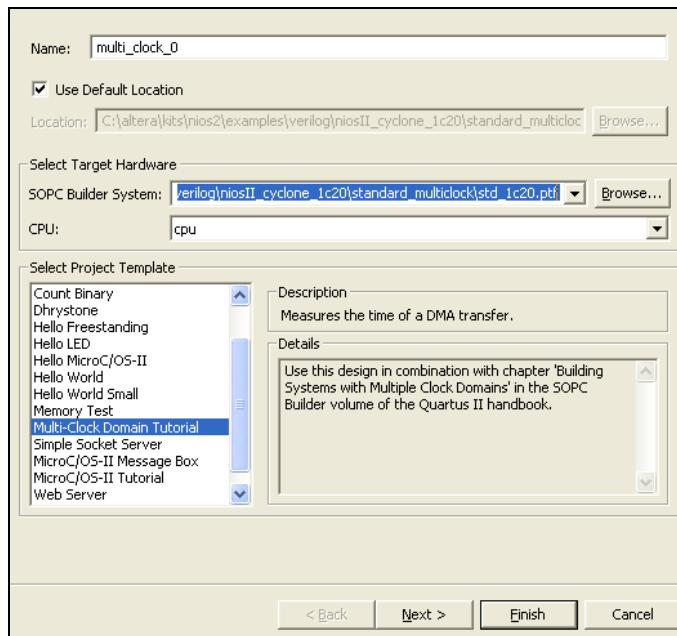
Create a New Nios II IDE Project

To create a new Nios II IDE project using the provided example files, perform the following steps:

1. Start the Nios II IDE.
2. Choose **New > C/C++ Application** (File menu) to start a new project. The first page of the **New Project** wizard appears.

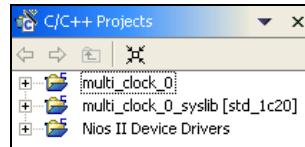
3. Under **Select Project Template**, select **Multi-Clock Domain Tutorial**.
4. Ensure that **Use Default Location** is turned on.
5. Click **Browse** under **Select Target Hardware**. The **Select Target Hardware** dialog box appears.
6. Browse to the *<hardware files directory>* directory where you created the hardware design earlier.
7. Select the file **std_<FPGA>.ptf**.
8. Click **Open** to return to the New Project wizard. The **SOPC Builder System** field is now specified and the **CPU** field now contains the name of the CPU in the system, as shown in [Figure 7–9](#).
9. Click **Finish**.

Figure 7–9. New Project Wizard Filled in with Correct Settings



After the IDE successfully creates the new project, the **C/C++ Projects** view contains two new projects, **multi_clock_0** and **multi_clock_0_syslib**, in addition to **Nios II Device Drivers**, as shown in Figure 7–10.

Figure 7–10. New Projects Displayed in the C/C++ Projects View



Build and Run the Program

In this section, you will build the software in the Nios II IDE and run it on a development board. In short, the program does the following:

- Initializes the read and write memory buffers, filling the read memory buffer with random values.
- Performs timer initialization to accurately measure how long a DMA transaction will take.
- Sets up a DMA transaction to copy the contents of the read buffer to the write buffer.
- Starts the timer, initiates the DMA transaction, and waits for the DMA to generate an interrupt.
- Stops the timer when the DMA transaction finishes.
- Verifies that the write buffer contents are correct.
- Reports the duration of the DMA transaction.

To build and run the program, perform the following steps:

1. In the Nios II IDE C/C++ Projects view, select the **multi_clock_0** project.
2. Choose **Run As > Nios II Hardware** (Run menu) to build the program, download it to the board, and run it. The IDE automatically builds the program before attempting to run it. The build process can take several minutes. After the build completes, the IDE will download the program to the target board and run it.
3. View the results in the **Console** view.

The **Console** view will display results similar to the following:

```
nios2-terminal: starting in terminal mode (Control-C  
exits)  
  
Hello from Nios II!  
Starting DMA transfer...  
  
Starting a DMA transfer of 4096 bytes of data.  
DMA transfer completed.  
It took 34.3600006104 useconds to complete the  
transfer.  
Comparing send and receive buffer data...  
Data Matches.  
  
Program completed successfully.
```



See the Nios II IDE online help for more information on building and downloading projects.

In this example, the DMA achieved approximately 120 MBytes per second ($4096\text{ bytes} / 34.36\text{ usec}$). The source (read) and destination (write) buffer memories have zero wait states, and therefore can perform a maximum of one transfer per clock cycle. For successive 32-bit transfers at 100 MHz, the theoretical maximum DMA transfer performance is 400 MBytes per second. The 34.36 usec time includes the following processor overhead, which accounts for the difference between 400 and 120:

- Time spent in the DMA driver setting up the DMA transfer.
- Time spent in the interrupt handler after the DMA flags that it has completed the transfer.
- Time spent entering the DMA callback function to capture the finish time.

Conclusion

Congratulations! You have completed all tutorial steps in this chapter to create a multi-clock domain system with SOPC Builder and exercise the system in hardware.

A

Adding a Component to the System 2–4
Address Decoding 3–4
Arbitration for Multi-Master Systems 3–11
 Details 3–14
 Fairness-Based Shares 3–15
 Round-Robin Scheduling 3–16
 Rules 3–15
 Setting Parameters in the SOPC Builder
 GUI 3–17
 Slave-Side Arbitration 3–13
 Traditional Shared Bus Architectures 3–12
Architecture of SOPC Builder Systems 1–1
Avalon Switch Fabric 1–5
 Functions 3–3
 High-Level Description 3–1
 Implementation & Fundamentals 3–3
 Interface 6–13

C

Clock
 Domain Assignments 7–7
 Domain Crossing 3–18
 Domain Crossing Logic 3–18, 3–20
 Duration of Transfers Crossing
 Domains 3–20
Example Design Overview 7–1
Implementing Multiple Domains in the SOPC
 Builder GUI 3–21

Component 1–2
 Development Flow 6–3
 Hardware Design 6–4

 Directory
 hdl Directory 4–5
 Directory Location 4–6
 Files, Other 4–5
 Hardware 4–2
 Identifying Information 5–8
Include Logic Inside the System Module 4–2
Interface to Logic Outside the System

Module 4–3
Parameters 5–9
Re-Editing 5–11
Sharing 6–28
Specifying Connections, Base Address, Clock
 & IRQ 2–5
System-Level Verification 6–7
Table of Active Components 2–7
User Defined Components 1–4, 2–7
Wizard Tab 5–8, 6–20
Component Editor 6–1
 HDL Files Tab 5–3, 6–15
 Interfaces Tab, SOPC Builder GUI 5–5
 Output 5–2
 Signals Tab 5–4, 6–17
 SW Files Tab 5–6
Component, Sources 4–1
Components
 class.ptf File 4–3
 Scripts
 cb_generator.pl File 4–4
Connection Panel 2–6

D

Data-Path Multiplexing 3–5
Design Example
 Pulse-Width Modulator Slave (PWM) 6–8
Dynamic Bus Sizing 3–10

F

Fairness-Based Shares 3–15

I

Interrupt Controller 3–21
 Assigning IRQs in the SOPC Builder
 GUI 3–23
 Hardware Priority 3–23
 Software Priority 3–22

L

Latency Capabilities [3–7](#)

M

Memory Map for Software Development [1–6](#)

Minimum Share Value [3–17](#)

Multi-Clock Hardware System [7–3](#)

N

Narrower Master [3–11](#)

Native Address Alignment [3–9](#)

New Nios II IDE Project [7–16](#)

P

PWM Design Specifications [6–9](#)

PWM Example Design

 Design Files [6–10](#)

 Quartus II Design to Use the PWM

 Output [6–23](#)

 Software API [6–14](#)

 Task Logic [6–11](#)

R

Register File [6–12](#)

Reset Distribution [3–24](#)

S

SDK Option, SOPC Builder [2–11](#)

Simulation Model & Testbench [1–6](#)

SOPC Architecture [1–1](#)

SOPC Builder

 Defining & Generating the System

 Hardware [1–5](#)

 Functions [1–5](#)

 HDL Option [2–11](#)

 Ready Components [1–3](#)

 SDK Option [2–11](#)

 Setup Preferences [2–13](#)

 Simulation Option [2–12](#)

 Starting [2–1](#)

 Starting a New System [2–1](#)

 Starting the Component Editor [5–3](#)

 Structure & Contents of a Component

 Directory [4–3](#)

 System Contents Tab [2–3](#)

 System Dependency Tabs [2–8](#)

 System Generation [2–13](#)

 System Generation Tab [2–9, 2–10](#)

 Sources of Components [4–1](#)

T

Tools menu, Other [2–13](#)

W

Wait-State Insertion [3–6](#)

Wider Master [3–10](#)