



Quartus II Handbook, Volume 1

Design & Synthesis



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

qii5v1-2.1

Copyright © 2004 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. Design Flows

Revision History	Section I-2
------------------------	-------------

Chapter 1. Hierarchical Block-Based & Team-Based Design Flows

Introduction	1-1
--------------------	-----

Design Flows: Flattened versus Hierarchical Block-Based	1-1
---	-----

Block-Based & Team-Based Designs	1-2
--	-----

Block-Based Design with the Quartus II LogicLock Methodology	1-4
--	-----

Preserving Timing Results Using the LogicLock Flow	1-5
--	-----

Preserving Routing	1-6
--------------------------	-----

Design Partitioning & Creating Multiple Netlist Files	1-6
---	-----

Performing Incremental Fitting	1-8
--------------------------------------	-----

Save a Node-Level Netlist into a Persistent Source File (Verilog Quartus Mapping File). ...	1-8
---	-----

Prevent Further Netlist Optimization	1-9
--	-----

Conclusion	1-9
------------------	-----

Chapter 2. Quartus II Design Flow for MAX+PLUS II Users

Introduction	2-1
--------------------	-----

Chapter Overview	2-1
------------------------	-----

Typical Design Flow	2-2
---------------------------	-----

Device Support	2-3
----------------------	-----

Quartus II GUI Overview	2-4
-------------------------------	-----

Project Navigator	2-4
-------------------------	-----

Node Finder	2-4
-------------------	-----

Tcl Console	2-4
-------------------	-----

Messages	2-4
----------------	-----

Status	2-5
--------------	-----

Setting up MAX+PLUS II Look and Feel in Quartus II	2-6
--	-----

Compiler Tool	2-8
---------------------	-----

Converting an Existing MAX+PLUS II Design	2-10
---	------

Converting MAX+PLUS II Graphic Design Files	2-11
---	------

Importing MAX+PLUS II Assignments	2-12
---	------

Quartus II Design Flow	2-13
Creating a New Project	2-14
Design Entry	2-14
Making Assignments	2-17
Synthesis	2-20
Functional Simulation	2-20
Place & Route	2-22
Timing Analysis	2-23
Timing Closure Floorplan	2-25
Timing Simulation	2-26
Power Estimation	2-28
Programming	2-29
Conclusion	2-29
Quick Menu Reference	2-30

Chapter 3. System Design Using SOPC Builder

Introduction	3-1
SOPC Builder Peripherals	3-2
Embedded Software Applications	3-4
Avalon Switch Fabric	3-4
System Generation	3-6
Simulation Model & Testbench	3-6
Using SOPC Builder	3-6
System Contents Page	3-7
System Generation Page	3-9
System Dependency Pages	3-12
Generating a System	3-13
Further Information	3-13

Chapter 4. Quartus II Support for HardCopy Devices

Introduction	4-1
Features	4-2
HARDCOPY_FPGA_PROTOTYPE, HardCopy Stratix, and Stratix Devices	4-3
HardCopy Design Flow	4-4
The Design Flow Steps of the One Step Process	4-6
How to Design HardCopy Devices	4-6
Targeting Designs to HARDCOPY_FPGA_PROTOTYPE Devices	4-6
Tcl Support for HardCopy Migration	4-9
Design Optimization & Performance Estimation	4-10
HardCopy Floorplans & Timing Models	4-10
Performance Estimation	4-10
Placement Constraints	4-12
Location Constraints	4-13
LAB Assignments	4-13
LogicLock Assignments	4-14
Targeting Designs to HardCopy APEX 20KC and HardCopy APEX 20KE Devices	4-14
Checking Designs for HardCopy Design Guidelines	4-15

Design Assistant Settings	4-15
Running Design Assistant	4-15
Reports and Summary	4-15
Generating the HardCopy Design Database	4-16
Static Timing Analysis (STA)	4-17
Power Estimation	4-17
HardCopy Stratix Power Calculator	4-17
Opening HardCopy Stratix Power Calculator	4-18
HardCopy APEX 20K Power Calculator	4-20
Power Calculators for FPGAs	4-20
Tcl Support for HardCopy Stratix	4-20
Conclusion	4-20
Related Documents	4-21

Chapter 5. Engineering Change Management

Impact of Last Minute Design Changes	5-1
Performance	5-1
Compile Time	5-2
Verification	5-2
Documentation	5-2
ECO Support	5-2
ECO Support at the HDL Level	5-3
ECO Support at the Netlist Level	5-5
Conclusion	5-6

Section II. Design Guidelines

Revision History	Section II-1
------------------------	--------------

Chapter 6. Design Recommendations for Altera Devices

Introduction	6-1
Synchronous FPGA Design Practices	6-1
Fundamentals of Synchronous Design	6-2
Hazards of Asynchronous Design	6-2
Recommended Design Techniques	6-3
Combinational Logic Structures	6-4
Clocking Schemes	6-7
Hierarchical Design Partitioning	6-12
Targeting Clock & Register-Control Architectural Features	6-14
Conclusion	6-15

Chapter 7. Recommended HDL Coding Styles

Introduction	7-1
Instantiating and Inferring Altera Megafunctions	7-1
Instantiating Altera Megafunctions in HDL Code	7-2
Inferring Megafunctions from HDL Code	7-4

Counters	7-5
Adder/Subtractors	7-6
Multipliers	7-8
Multiply-Accumulators & Multiply-Adders	7-10
RAM	7-14
ROM	7-20
Shift Registers	7-21
Device-Specific Coding Recommendations	7-25
Secondary Control Signals in Registers or Flip-Flops	7-25
Tri-State Signals	7-27
Adder Trees	7-28
General Coding Recommendations	7-31
Latches	7-31
State Machines	7-32
..... Multiplexers	7-38
Conclusion	7-47

Section III. Synthesis

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

Chapter 8. Quartus II Integrated Synthesis

Introduction	8-1
Verilog HDL & VHDL Support	8-1
Verilog HDL	8-1
VHDL	8-2
Types of Synthesis Options	8-3
Synthesis Directives	8-4
Synthesis Attributes	8-5
Quartus II Logic Options	8-6
Quartus II Synthesis Options	8-6
Translate Off & On	8-7
Read Comments as HDL	8-7
Full Case	8-8
Parallel Case	8-9
Keep Combinational Node/Implement as Output of Logic Cell	8-10
Preserve Registers	8-11
Maximum Fan-Out	8-12
Optimization Technique	8-13
State Machine Processing	8-14
Preserve Hierarchical Boundary	8-15
Restructure Multiplexers	8-16
Power-Up Level	8-18
Power-Up Don't Care	8-19
Remove Duplicate Logic	8-19
Remove Duplicate Registers	8-20
Remove Redundant Logic Cells	8-20

Megafunction Inference Control	8–20
RAM Style	8–22
Setting Other Quartus II Options in Your HDL Source Code	8–23
Use I/O Flip-Flops	8–23
Altera Attribute	8–25
Chip Pin	8–27
Scripting Support	8–29
Quartus II Synthesis Options	8–29
Assigning a Pin	8–31
Conclusion	8–31

Chapter 9. Synplicity Synplify & SynplifyPro Support

Introduction	9–1
Design Flow	9–1
Synplify Optimization Strategies	9–6
Implementations in Synplify Pro	9–6
Timing-driven Synthesis Settings	9–6
Finite State Machine (FSM) Compiler	9–9
General Optimization Attributes & Options	9–10
Altera Specific Attributes	9–11
Exporting Designs to the Quartus II Software Using NativeLink Integration	9–13
Running the Quartus II Software from within the Synplify Software	9–14
Using the Quartus II Software to Launch the Synplify Software	9–14
Cross-Probing with the Quartus II Software	9–15
Enabling Cross-Probing	9–15
Cross-Probing from the Quartus II Software	9–16
Cross-Probing from the Synplify Software	9–16
Guidelines for Altera Megafunctions & Architecture-Specific Features	9–17
Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager	9–18
Inferring Altera Megafunctions from HDL Code	9–23
Hierarchy & Design Considerations with Multiple VQM Files	9–29
Creating a Design with Multiple VQM Files	9–29
Creating a Design with Multiple VQM Files using Multipoint Synthesis (Synplify Pro only) ...	9–30
Generating a Design with Multiple VQM Files Using Black Boxes	9–36
Conclusion	9–41

Chapter 10. Mentor Graphics LeonardoSpectrum Support

Introduction	10–1
Design Flow	10–1
Optimization Strategies	10–4
Timing-Driven Synthesis	10–4
Other Constraints	10–5
Timing Analysis with the Leonardo-Spectrum Software	10–7
Exporting Designs Using NativeLink Integration	10–8
Generating Netlist Files	10–8

Including Design Files for Black-Boxed Modules	10-8
Passing Constraints Via Scripts	10-8
Integration with the Quartus II Software	10-9
Guidelines for Altera Megafunctions & LPM Functions	10-9
Inferring Multipliers & DSP Functions	10-11
Controlling DSP Block Inference	10-12
Block-based Design with the Quartus II LogicLock Methodology	10-18
Hierarchy & Design Considerations	10-19
Creating a Design with Multiple EDIF Files	10-20
Generating Multiple EDIF Files Using Black Boxes	10-24
Incremental Synthesis Flow	10-29
Conclusion	10-31

Chapter 11. Mentor Graphics Precision RTL Synthesis Support

Introduction	11-1
Design Flow	11-1
Creating a Project & Compiling a Design	11-5
Creating a Project	11-5
Compiling the Design	11-6
Setting Constraints	11-6
Setting Timing Constraints	11-7
Setting Mapping Constraints	11-7
Assigning Pin Numbers & I/O Settings	11-8
Assigning I/O Registers	11-9
Disabling I/O Pad Insertion	11-9
Controlling Fan-Out on Data Nets	11-10
Synthesizing the Design & Evaluating the Results	11-11
Obtaining Accurate Logic Utilization & Timing Analysis Reports	11-11
Megafunctions & Architecture-Specific Features	11-14
Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager	11-15
Inferring Altera Megafunctions from HDL Code	11-17
Block-Based Design with the Quartus II LogicLock Methodology	11-23
Hierarchy & Design Considerations	11-23
Creating a Design with Separate Blocks for the LogicLock Methodology	11-24
Creating a Design with Separate Blocks Using the LogicLock Attribute in a Single Precision Project	11-25
Generating a Design with Multiple EDIF Files Using Black Boxes	11-26
Conclusion	11-30

Chapter 12. Synopsys FPGA

Compiler II BLIS & Quartus II LogicLock Design Flow

Introduction	12-1
Design Hierarchy	12-1
Block-Level Incremental Synthesis	12-2
FPGA Compiler II Design Block	12-2
FPGA Compiler II & Quartus II Synthesis	12-3
Block Root	12-3
How the BLIS Feature Works with the LogicLock Feature	12-4

Hierarchy Considerations	12-5
Time Stamp Synthesis	12-6
Creating & Maintaining a Design	12-6
Opening the Modules Constraint Table & Labeling Block Roots	12-7
Exporting Block-Level Netlist Files	12-7
Changing Source Within a Block	12-8
Removing a Block Root	12-9
Using BLIS Shell Commands	12-9
Conclusion	12-10

Chapter 13. Synopsys Design Compiler FPGA Support

Design Flow Using the DC FPGA Software & the Quartus II Software	13-2
Setup of the DC FPGA Software Environment for Altera Device Families	13-3
Megafunctions & Architecture-Specific Features	13-5
Reading MegaWizard-Generated Variation Wrapper Files	13-7
Using MegaWizard-Generated Variation Wrapper Files in a Black-Box Methodology	13-7
Inferring Altera Megafunctions from HDL Code	13-8
Reading Design Files into the DC FPGA Software	13-9
Selecting a Target Device	13-11
Compilation & Synthesis	13-14
Saving Synthesis Results	13-17
Exporting Designs to the Quartus II Software	13-18
Place & Route with the Quartus II Software	13-21
Conclusion	13-21

Chapter 14. Analyzing Designs with the Quartus II RTL Viewer & Technology Map Viewer

Introduction	14-1
RTL Viewer Overview	14-1
Technology Map Viewer Overview	14-2
Quartus II Design Flow with the RTL & Technology Map Viewers	14-3
Introduction to the User Interface	14-4
Schematic View	14-5
Hierarchy List	14-12
Navigating the Schematic View	14-13
Zooming & Magnification	14-13
Page Partitioning in the Schematic View	14-14
Traversing the Design Hierarchy	14-16
Back & Forward Page Viewing	14-17
Go to Net Driver	14-17
Filtering in the Schematic View	14-17
Examples of Filtered Netlists	14-19
Expanding a Filtered Netlist	14-21
Reducing a Filtered Netlist	14-21
Probing to Source Design File & Other Quartus II Features	14-22
Viewing a Timing Path in the Technology Map Viewer	14-22
Other Features in the Schematic Viewer	14-24
Tooltips	14-24

Displaying Net Names	14-26
Full Screen View	14-26
Find Command	14-26
Exporting Schematic as JPEG or BMP Image & Copying to Clipboard	14-27
Printing	14-28
Using the RTL & Technology Map Viewers to Analyze Design Problems	14-28
Conclusion	14-29
Index	1



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. Hierarchical Block-Based & Team-Based Design Flows

Revised: *August 2004*

Part number: *qii51001-2.1*

Chapter 2. Quartus II Design Flow for MAX+PLUS II Users

Revised: *June 2004*

Part number: *qii51002-2.0*

Chapter 3. System Design Using SOPC Builder

Revised: *June 2004*

Part number: *qii51003-2.0*

Chapter 4. Quartus II Support for HardCopy Devices

Revised: *June 2004*

Part number: *qii51004-2.0*

Chapter 5. Engineering Change Management

Revised: *June 2004*

Part number: *qii51005-2.0*

Chapter 6. Design Recommendations for Altera Devices

Revised: *June 2004*

Part number: *qii51006-2.0*

Chapter 7. Recommended HDL Coding Styles

Revised: *June 2004*

Part number: *qii51007-2.0*

Chapter 8. Quartus II Integrated Synthesis

Revised: *June 2004*

Part number: *qii51008-2.0*

Chapter 9. Synplicity Synplify & SynplifyPro Support

Revised: *June 2004*

Part number: *qii51009-2.0*

Chapter 10. Mentor Graphics LeonardoSpectrum

Support

Revised: *June 2004*Part number: *qii51010-2.0*

Chapter 11. Mentor Graphics Precision RTL Synthesis Support

Revised: *June 2004*Part number: *qii51011-2.0*

Chapter 12. Synopsys FPGA

Compiler II BLIS & Quartus II LogicLock Design Flow

Revised: *June 2004*Part number: *qii51012-1.0*

Chapter 13. Synopsys Design Compiler FPGA Support

Revised: *June 2004*Part number: *qii51014-1.0*

Chapter 14. Analyzing Designs with the Quartus II RTL Viewer & Technology Map Viewer

Revised: *June 2004*Part number: *qii51013-2.0*



About this Handbook

This handbook provides comprehensive information about the Altera® Quartus®II design software, version 4.0..

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	lit_req@altera.com (1)	lit_req@altera.com (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} , lqdesigns 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>.pdf 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.
	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 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 options that are available for each of these flows.

This section includes the following chapters:

- Chapter 1, Hierarchical Block-Based & Team-Based Design Flows
- Chapter 2, Quartus II Design Flow for MAX+PLUS II Users
- Chapter 3, System Design Using SOPC Builder
- Chapter 4, Quartus II Support for HardCopy Devices
- Chapter 5, Engineering Change Management

Revision History

The table below shows the revision history for [Chapters 1 to 5](#).

Chapter(s)	Date / Version	Changes Made
1	Aug. 2004 v2.1	<ul style="list-style-type: none"> • Minor typographical corrections
	June 2004 v2.0	<ul style="list-style-type: none"> • Updates to tables, figures. • New functionality for Quartus 4.1.
	Feb. 2004 v1.0	Initial release.
2	June 2004 v2.0	<ul style="list-style-type: none"> • Updates to tables, figures. • New functionality for Quartus 4.1.
	Feb. 2004 v1.0	Initial release.
3	June 2004 v2.0	<ul style="list-style-type: none"> • Updates to tables, figures. • New functionality for Quartus 4.1.
	Feb. 2004 v1.0	Initial release.
4	June 2004 v2.0	<ul style="list-style-type: none"> • Updates to tables, figures. • New functionality for Quartus 4.1.
	Feb. 2004 v1.0	Initial release.
5	June 2004 v2.0	No change to document.
	Feb. 2004 v1.0	Initial release.

Introduction

Today's complex designs require multiple hardware description language (HDL) design files, each of which may undergo significant testing and optimization before being combined into the final top-level design. Many designs require work from more than one member of a design team. In this environment, the traditional flattened netlist approach to design may not be as effective as a hierarchical block-based design methodology.

This chapter discusses the differences between flattened and hierarchical design flows and describes block-based or team-based hierarchical methodologies in detail. The chapter highlights the Altera® Quartus® II LogicLock™ design methodology, and discusses issues to consider when partitioning a design to achieve optimal results when using this methodology.

Design Flows: Flattened versus Hierarchical Block-Based

Most HDL-based designs are created using either a block-based or a flattened design methodology. In a flattened synthesis flow, you apply a single set of optimizations to the design's top level. Thus, a flattened design has one output netlist file for the entire design. However, as designs become more complex and designers work in teams, a block-based hierarchical design flow is often more effective. In this approach, each sub-block may have its own output netlist file and you perform optimization on individual sub-blocks. After you optimize all of the sub-blocks, you integrate them into a final design and can optimize it at the top level if desired. Synthesizing and optimizing each sub-block separately may provide better quality of results.

Using a block-based design methodology can also reduce the placement and routing changes required with each compilation in the Quartus II software. Using a hierarchical design approach limits the amount of logic impacted by engineering change orders (ECOs) that affect only one part of the design.

When you make small changes to a design, you can use incremental fitting for Stratix® II, Stratix, Stratix GX, Cyclone™, or MAX® II devices by choosing **Start > Start Incremental Fitting** (Processing menu). Incremental fitting updates the design's netlist, placement, and routing, while ensuring that the timing characteristics of the design change as little as possible from those of the previous compilation. Incremental

fitting also helps to reduce the compilation time necessary to regenerate a netlist. If the changes to the design are too big, complexities in incremental fitting may cause longer compilation times.



For more information on incremental fitting and the circumstances under which it can be used, refer to the Quartus II Help.

When you make changes to a single block in the design, you can use the LogicLock design methodology to preserve your performance results, as discussed in “[Block-Based Design with the Quartus II LogicLock Methodology](#)” on page 1-4.

Table 1-1 describes each design flow and its advantages.

Table 1-1. Quartus II Flattened Versus Block-Based Hierarchical Design Flow		
Design Flow	Description	Advantages
Traditional flattened	One output netlist for the entire design	<ul style="list-style-type: none"> You can perform optimization across design boundaries and hierarchies for the entire design. Simple to manage.
Block-based hierarchical	Separate netlist files for design modules	<ul style="list-style-type: none"> You compile each module separately. You can apply different optimization techniques to each module. Design modifications do not affect the optimization of other modules if the placement of other modules is locked down in the device. You can use optimized modules in other designs.

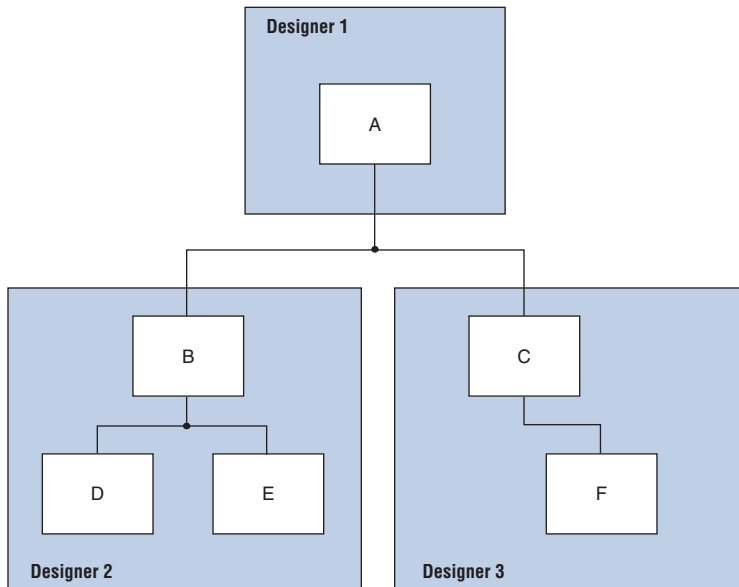
Block-Based & Team-Based Designs

For larger designs, such as those implemented in today’s large high performance devices, a team of designers may work on different modules of a design at the same time.

To take advantage of a block-based design flow, you must define different modules as a part of your design hierarchy in different files and instantiate them in a top-level file.

Figure 1–1 shows an example of a design hierarchy.

Figure 1–1. Quartus II Design Hierarchy



In Figure 1–1, 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.

You can treat each module or a group of modules as one block of the design for block-based synthesis. A submodule can be a Verilog HDL module, a VHDL entity, an ADHL (.tdf) submodule, a Block Design File (.bdf) entity, a Verilog Quartus Mapping (.vqm), Electronic Data Interchange Format (.edf) netlist file, or any combination of these. During synthesis, you generate a separate VQM or EDF netlist file for each block of submodules. In this case, there is a separate netlist file for each block including modules A, B, and C.

To combine these submodules into a block for synthesis, they must form a single tree in the hierarchical design. For example, you cannot create one netlist file for the two submodules E and C, while A and B are in different netlists, because E and C are in different branches of the design. You can have E and C separate with individual netlists for A, B, C, and E, or have E and C grouped in one netlist for the whole tree under the top-level design A.

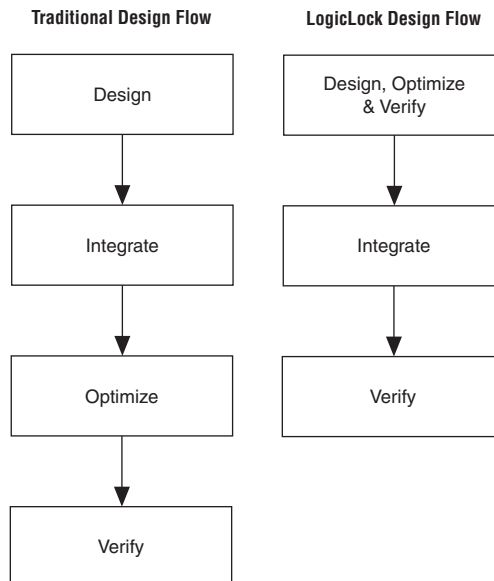
Block-Based Design with the Quartus II LogicLock Methodology

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, then 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 1–2 compares the traditional design flow with the LogicLock design flow.

Figure 1–2. Comparison of Traditional Design Flow with Quartus II LogicLock Design Flow





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*. The rest of this chapter assumes that you are familiar with the basic LogicLock features and methodology.

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 with 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 **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 imported.

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 different 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 if 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.

Preserving Routing

LogicLock regions not only allow you to preserve logic placement from one compilation to the next, but they also allow you to retain routing inside LogicLock regions. You can back-annotate and export the routing of a submodule, then import it into a top-level project. This feature allows you to specify the exact location of the submodule in the device and which routing resources the Quartus II PowerFit™ Fitter should use during compilation.

You can import back-annotated routing if exactly one instance of the imported region exists in the top level of the design. If more than one instance exists, the routing constraint is ignored and the LogicLock region is imported without back-annotation of routing. The routing constraint cannot be applied to multiple instances in different parts of the device because routing channels from one part of the device may not be exactly the same in another area of the device.



For more information on back-annotating routing, see the *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*.

Design Partitioning & Creating Multiple Netlist Files

When using a block-based design methodology, you typically create separate netlists for separate design modules. Partitioning your design up-front is the best way to employ team-based methodologies and facilitate design reuse. In addition, to take advantage of the LogicLock design methodology when synthesizing a design using the Quartus II software, you should create an atom netlist for each design block before you lock down the nodes in that block into LogicLock regions.

When creating separate netlist files for a block-based methodology, it is important to consider how your design is partitioned into sub-blocks that will be separate netlists in your block-based methodology. Altera recommends using registered boundaries for all modules. This helps ensure that the timing between hierarchical blocks does not become the critical timing path in the design once the blocks are assembled.



See the *Design Recommendations for Altera Devices* chapter in Volume 1 of the *Quartus II Handbook* for more design partitioning guidelines.

Certain third-party synthesis tools allow you to create separate netlist files for different sections of a design hierarchy, or to maintain separate partitions within one netlist for different sections of a design hierarchy. To ensure that the synthesis tool functions properly, tools allow you to create separate netlist files or partitions only for blocks that contain entire 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 blocks, it is difficult to perform incremental synthesis. In this case, both regions must be recompiled when you change one of the modules or entities.

If you don't use a synthesis tool feature to automatically create separate netlist files, you can create a black box for each submodule in the higher-level file that instantiates it. Create a black box by first instantiating the submodule in the top-level design, then providing a component declaration in VHDL or a dummy module declaration in Verilog HDL. When creating a black box, you do not provide the actual design or logic that forms that submodule. You then create a netlist file for the submodule in a separate synthesis project. Essentially, you instantiate a wrapper for the submodule netlist in the top-level design or any higher module that instantiates it. Some synthesis tools have attributes that can be set to tell the synthesis tool that a submodule contained in a black box is intended to be empty.



See the appropriate chapter in the *Synthesis* section in Volume 1 of the *Quartus II Handbook* for details on your synthesis tool's support for creating multiple netlist files to be used with the LogicLock design methodology, and for more information on creating submodules contained in a black box.

If you synthesize a design using Quartus II Integrated Synthesis that contains a VHDL Design File (.vhd), Verilog Design File (.v), Text Design File (.tdf), or a Block Design File (.bdf), you must also create an atom netlist to establish fixed nodes and node names when using the LogicLock design methodology. Turn on the **Save a node-level netlist into a persistent source file (Verilog Quartus Mapping File)** option on the **Compilation Process** 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 in the **atom_netlists** directory under the current project directory. To create a different VQM with different Quartus II settings, change the file name setting on the **Compilation Process** page in the **Settings** dialog box (Assignments menu).



If you are using an atom netlist from a third-party synthesis tool and the design has black-boxed library of parameterized modules (LPM) functions or Altera megafunctions, you must generate a separate Quartus II VQM for the modules contained in the black box.

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 of the *Quartus II Handbook*.

Performing Incremental Fitting

You can perform incremental fitting with a Tcl command or with a command run at a command prompt.

Tcl Script

Use the following in a script or Tcl console:

```
execute_flow -incremental_fitting
```

The `execute_flow` command is in the `flow` package.

Command Prompt

Use the following at a system command prompt:

```
quartus_sh --flow incremental_fitting <project name> ↵
```

For more information about performing incremental fitting, see [page 1–1](#).

Save a Node-Level Netlist into a Persistent Source File (Verilog Quartus Mapping File).

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.

Prevent Further Netlist Optimization

Use the following Tcl statements to prevent further netlist optimization of nodes in a back-annotated LogicLock region:

```
foreach node [get_logiclock_contents \
-region <region name> -node_location] {
    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.

For more information about preventing further netlist optimization, refer to [“Preserving Timing Results Using the LogicLock Flow”](#) on page 1–5.

Conclusion

Hierarchical design methodologies can improve the efficiency of your design process, providing better design reuse opportunities and fewer integration problems when working in a team environment. Following the guidelines in this chapter can help you achieve good results with these methodologies.



2. Quartus II Design Flow for MAX+PLUS II Users

qii51002-2.0

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 the ability to choose an option in the Quartus II software to cause the graphical user interface (GUI) to display menus, tool bars, and utility windows as they appear in the MAX+PLUS II software without sacrificing functionality.

Chapter Overview

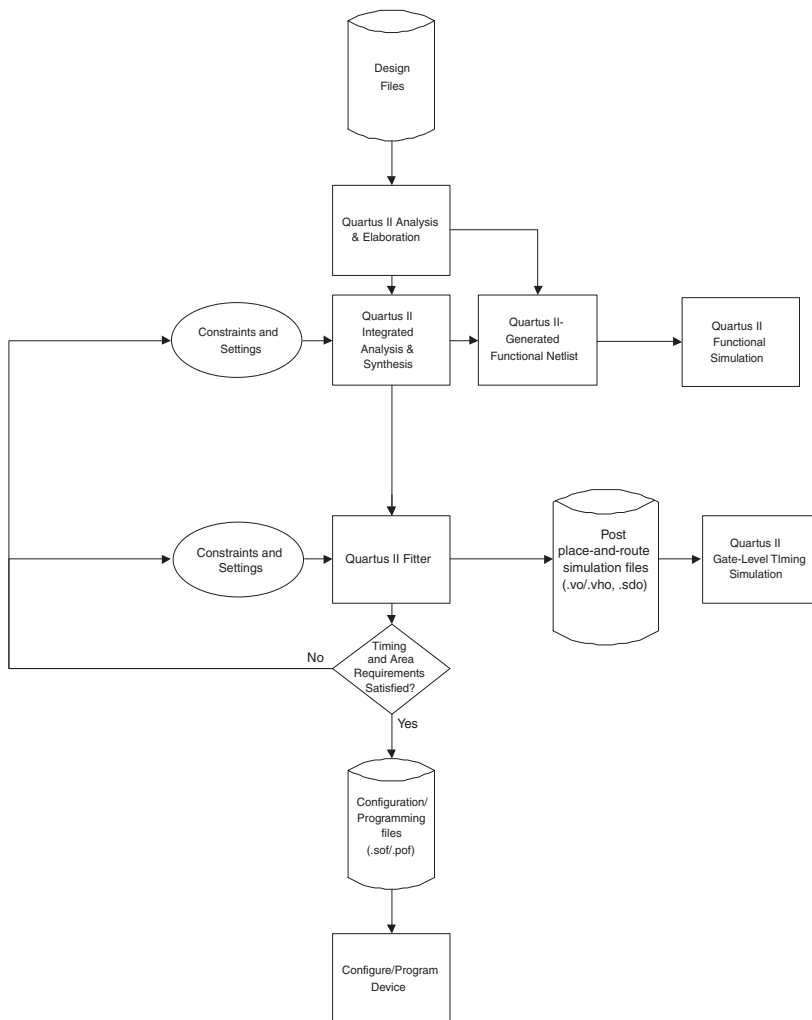
This chapter covers the following topics:

- Typical complex programmable logic device (CPLD) design flow
- Device support
- Quartus II GUI overview
- Setting up the MAX+PLUS II look and feel in the Quartus II software
- MAX+PLUS II look and feel
- Compiler tool
- MAX+PLUS II design conversion
- Quartus II design flow

Typical Design Flow

Figure 2–1 shows a typical design flow with the Quartus II software.

Figure 2–1. 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 2-1](#).

Table 2-1. Device Support Comparison		
Device Supported	Quartus II	MAX+PLUS II
Classic™		✓
MAX 3000A	✓	✓
MAX 7000S/AE/B	✓	✓
MAX 7000 /E		✓
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™	✓	
MAX II	✓	

Notes to Table 2-1:

- (1) PGA packages (G) 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 more 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. From the Tcl Console window you can 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 time elapsed are shown for each stage of the compilation.

Change Manager

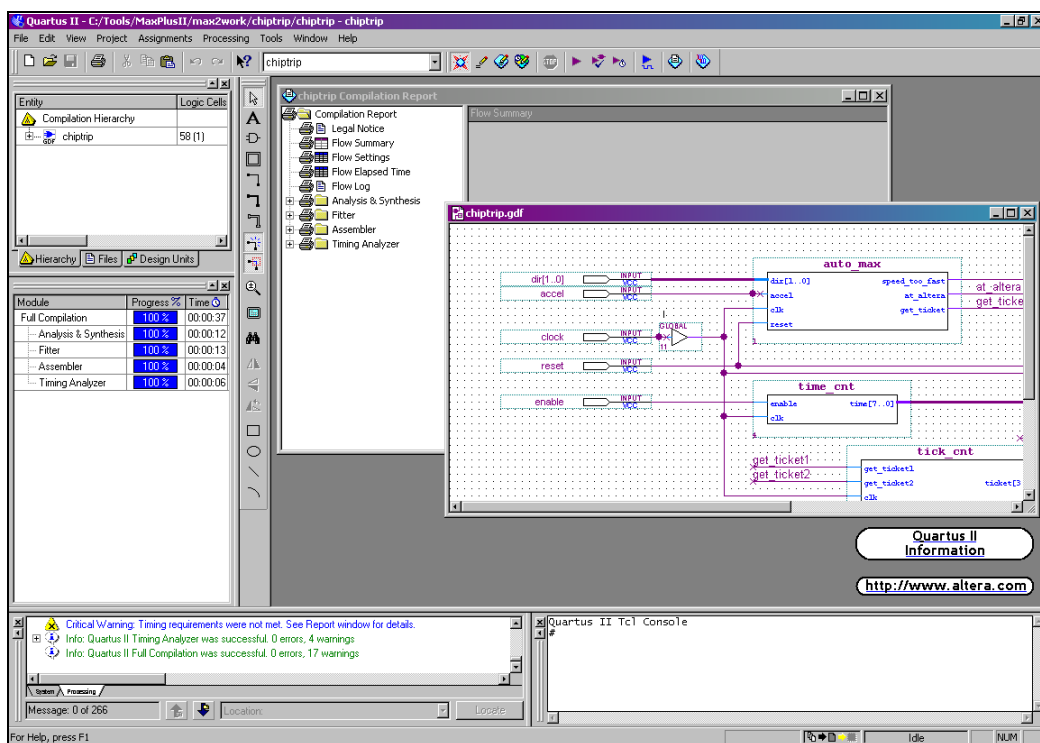
The Change Manager provides detailed tracking information on all design changes made with the Chip Editor.



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 2-2.

Figure 2-2. Example of the Quartus II Look and Feel

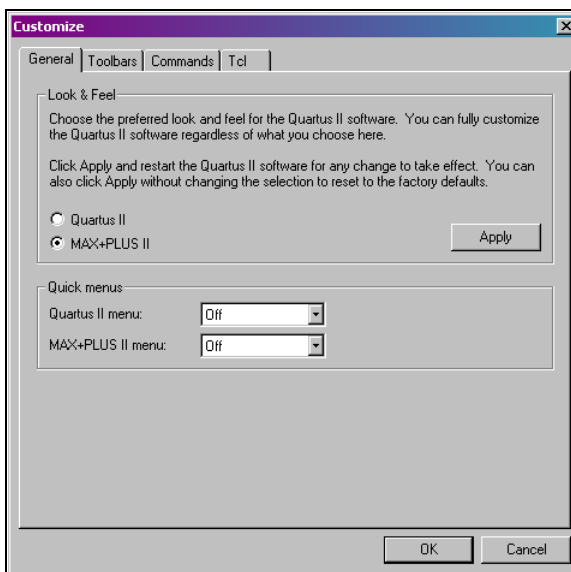


Setting up MAX+PLUS II Look and Feel in Quartus II

You can choose the MAX+PLUS II look and feel by selecting MAX+PLUS II in the **Look & Feel** box of the **General** tab of the **Customize** dialog box (Tools menu). Any changes to the look and feel does 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 on or off both Quartus II and MAX+PLUS II quick menus. 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). Click **Apply** without changing any of the selections if you want to restore the factory defaults (see [Figure 2-3](#)). Note: This was intended by design. If you simply click on Apply without changing anything, you will get the factory defaults.

Figure 2-3. Customize Dialog Box -- General Tab



MAX+PLUS II Look and Feel

The MAX+PLUS II look and feel of the Quartus II software closely resembles the MAX+PLUS II software. Figures 2–4 and 2–5 compare the appearance of the MAX+PLUS II look and feel.

Figure 2–4. MAX+PLUS II Software GUI

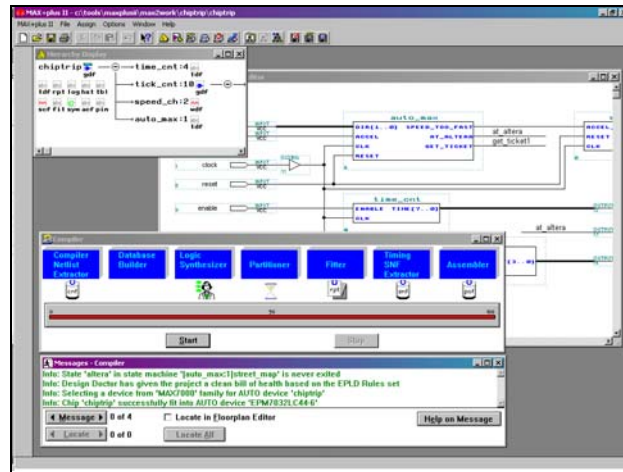
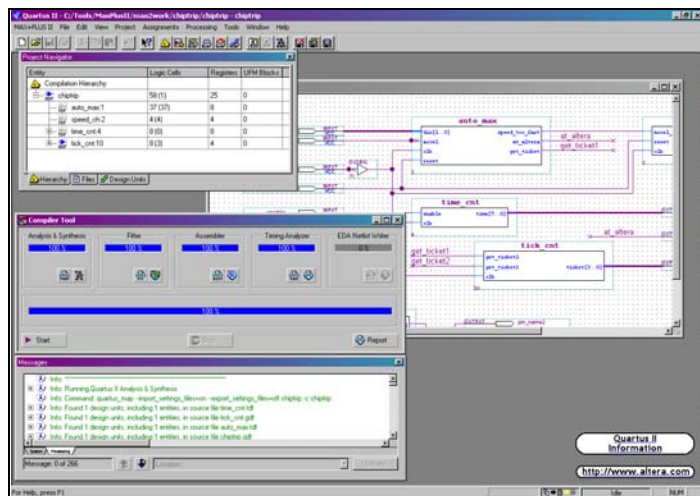


Figure 2–5. Quartus II Software with MAX+PLUS II Look & Feel



The standard MAX+PLUS II tool bar is also available with the MAX+PLUS II look and feel (see [Figure 2-6](#)).

Figure 2-6. 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:

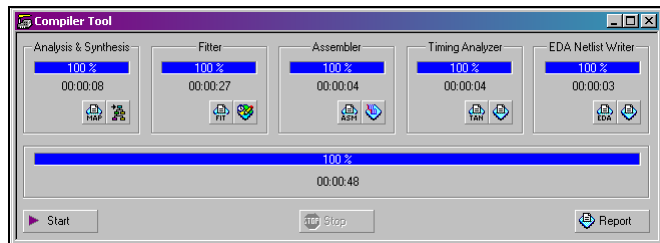
- Analysis and Synthesis
- Fitter
- Assembler
- Timing Analyzer
- EDA Netlist Writer

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 2-7](#)).



For information about Quartus II modules outside of the Compiler Tool, see the *Command-Line Scripting* chapter in Volume 2 of the *Quartus II Handbook*.

Figure 2-7. 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 are no modules 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 analogous 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 analogous with to 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

The Quartus II software can open and convert MAX+PLUS II designs and assignments. You can automatically convert an entire MAX+PLUS II design, or choose which assignments and files to convert.

The Quartus II software is project-based. All the files for your design (HDL input, simulation vectors, assignments etc.) are associated with a project file. For more information about creating a new project, see [“Creating a New Project” on page 2–14](#).

Converting an Existing MAX+PLUS II Design

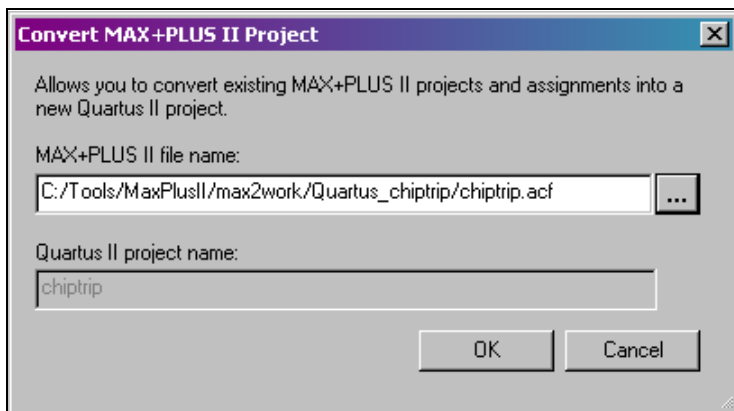
You can easily convert an existing MAX+PLUS II design for use with the Quartus II software with the **Open Project** (File menu) or **Convert MAX+PLUS II Project** (File menu) commands in the Quartus II software.

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. 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 (see [Figure 2–8](#)). 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 2–8. Convert MAX+PLUS II Design with Open Project Command

The conversion process performs the following actions:

- Converts the ACF into a QSF (equivalent to importing all MAX+PLUS II assignments)
- Creates a Quartus II Project File (.qpf)
- 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 GDF into a BDF using one of the following methods:

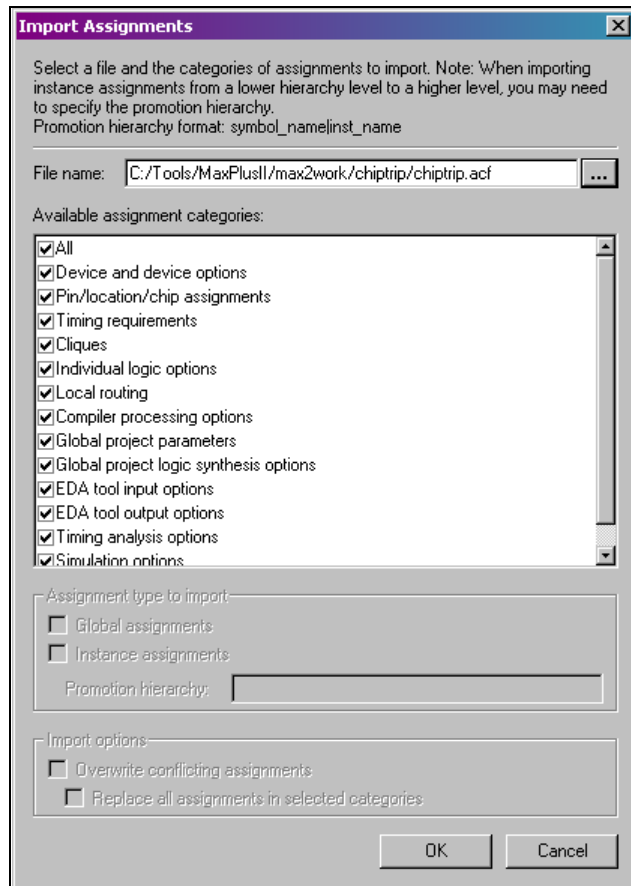
- Open the GDF and choose **Save As** (File menu). In the **Save As** dialog box, choose **Block Diagram/Schematic File (*.bdf)** from the **Save as** type list.
- Run the command line executable **quartus_g2b.exe** located in the *<Quartus II installation>/bin* directory. For example, to convert the 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 2-9](#)). You can also import QSF and ESF files.

Figure 2-9. 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 Quartus II and MAX+PLUS II software formats for node names and bus pin names are different. Make sure that the naming schemes map properly and do not interfere with design logic. Table 2-2 compares the differences between the naming conventions used by the Quartus II software and the MAX+PLUS II software.

Table 2-2. 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 `signal0` or `signal1`, 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 name after importing.

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.

To assist you through the Quartus II design flow, you can select from the following wizards to guide you through various settings:

- New project wizard
- Timing wizard
- Compiler settings wizard
- Simulator settings wizard
- Software build settings wizard

You can start the New Project Wizard from the File menu and the other wizards under **Wizards** (Assignments menu).

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)
- VHDL (.vhd)
- Verilog HDL (.v)
- Block Diagram File (.bdf)
- EDIF netlist file (.edf)
- VQM netlist file (.vqm)

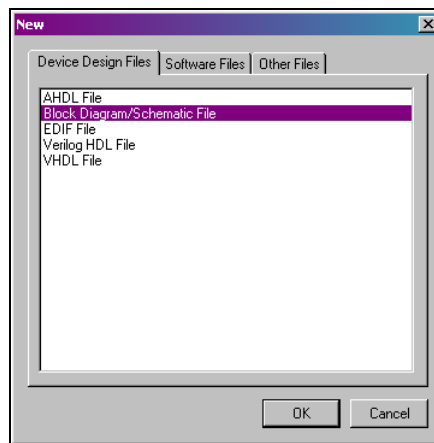
The Quartus II software has an advanced integrated synthesis engine that fully supports the Verilog HDL and VHDL languages and provides options to control the synthesis process.



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 2-10](#)).

Figure 2-10. New Dialog Box

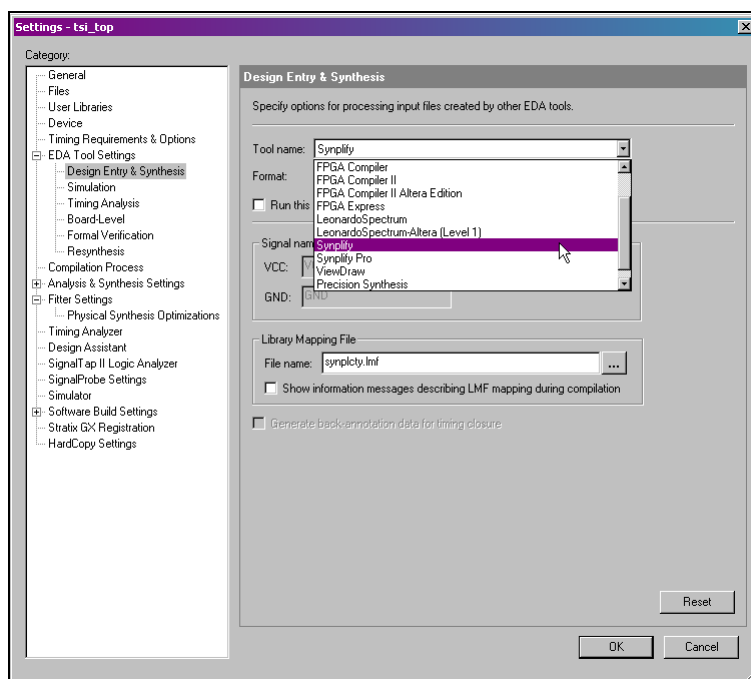




You can create other files, such as a Vector Waveform File (.vwf) from the **Software Files** tab and **Other Files** tab of the **New** dialog box (File menu).

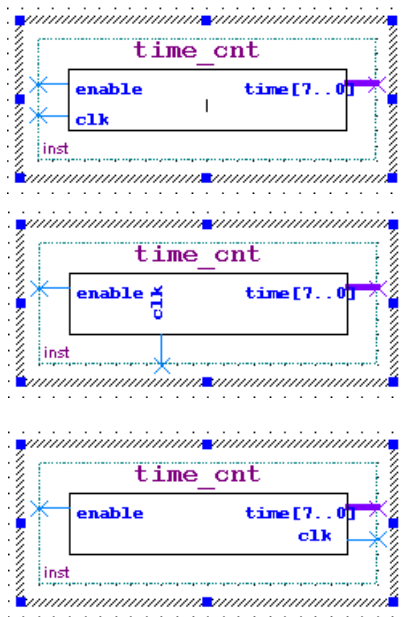
To analyze a netlist file created by an EDA tool, select the synthesis tool used to generate it in the **Tool** name list of the **Design Entry & Synthesis** page under **EDA Tool Settings** in the **Settings** dialog box (Assignments menu). See (Figure 2–11).

Figure 2–11. Settings Dialog Box



The Quartus II Block Editor has many advantages over the MAX+PLUS II Graphic Editor. The Block Editor offers an infinite amount of sheet space, multiple region selections, an enhanced Symbol Editor, and conduits.

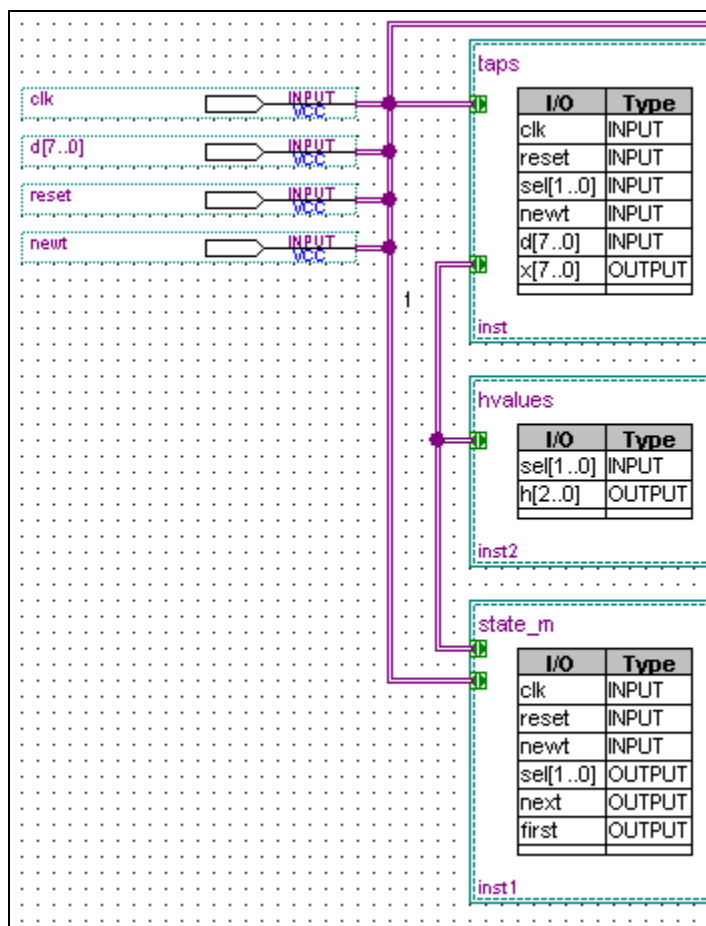
The Symbol Editor allows you to change the positions of the ports in a symbol (see the three images in Figure 2–12). You can reduce wire congestion around a symbol by changing the positions of the ports.

Figure 2–12. 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 2–13](#)). 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 2–13. Blocks and 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 has an intuitive spreadsheet interface designed to allow you to easily make, change, and manage a large number of 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 in the Assignment Editor:

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 with the **Timing Requirements & Options** page in 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.

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 2–14](#) shows a 4 ns t_{SU} assignment to a bus of registers made in the Assignment Editor.

Figure 2–14. Single T_{SU} Timing Assignment Applied to All Nodes of a Bus

	From	To	Assignment Name	Value
1	◆ *	◆ d[?]	tsu Requirement	4ns
2	<<new>>	<<new>>	<<new>>	



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

Synthesis

Quartus II integrated synthesis offers an alternative to EDA synthesis tools. Quartus II integrated synthesis fully supports VHDL and Verilog HDL synthesizable language features, as well as selected compiler directives.

You can set 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, APEX II, Cyclone, 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).

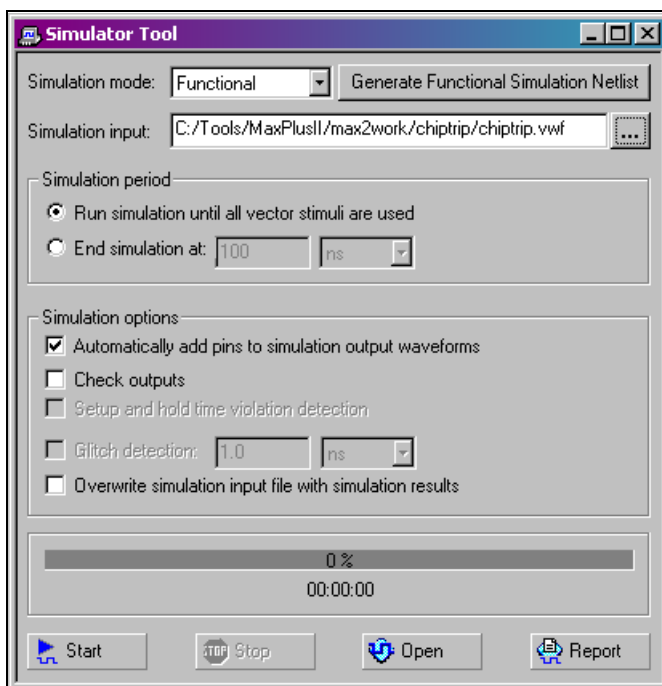


For more information, see the *Quartus II Integrated Synthesis* chapter in Volume 1 of the *Quartus II Handbook*.

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 2-15](#)) or choose **Generate Functional Simulation Netlist** (Processing menu).

Figure 2–15. Simulator Tool Ready for Functional Simulation

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

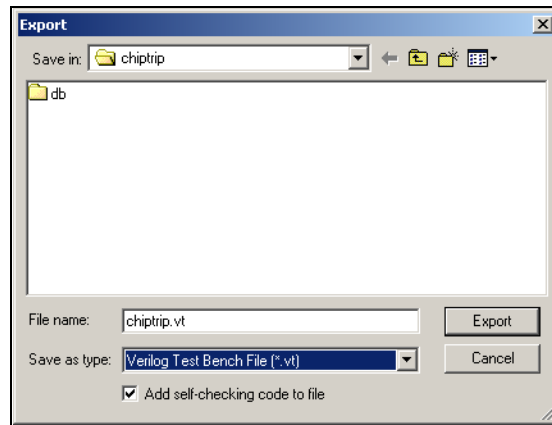
You can view and modify the simulator options on the **Simulator** page of the **Settings** dialog box or in the **Simulator Tool** window. You can set the simulation period and turn **Check outputs** on or off. You can choose to display the simulation outputs in the simulation report or in the vector waveform file (.vwf). To display the simulation results in the simulation input vector waveform file, turn on **Overwrite simulation input file with simulation results**.

When using either the MAX+PLUS II software or the 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 an 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 2-16](#)). Select **Verilog** or **VHDL testbench** from the **Save as type** list. Turn on **Add self-checking code to file** to add additional self-checking code to the testbench.

Figure 2-16. Export Dialog Box



Place & Route

The Quartus II Fitter, known as the PowerFit fitter, is the compiler module that fits your design into a device. The PowerFit fitter performs placement and routing.

You can turn on various fitter options located in the **Fitter Settings** page in 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. Quartus II has several options to reduce the time required to fit a design.

You can control the effort the Quartus II Fitter places by achieving 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 the length of time needed to compile is more important than achieving specific timing results, you can turn off these options.

You can control the amount of effort the Fitter makes by selecting **Standard Fit** or **Fast Fit**. Select **Standard Fit** in the **Fitter Effort** box of the **Fitter Settings** page in the **Settings** dialog box (Assignments menu) 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 the design's timing requirements. The **Auto Fit** option is available for Stratix II, Stratix GX, Stratix, and Cyclone devices.

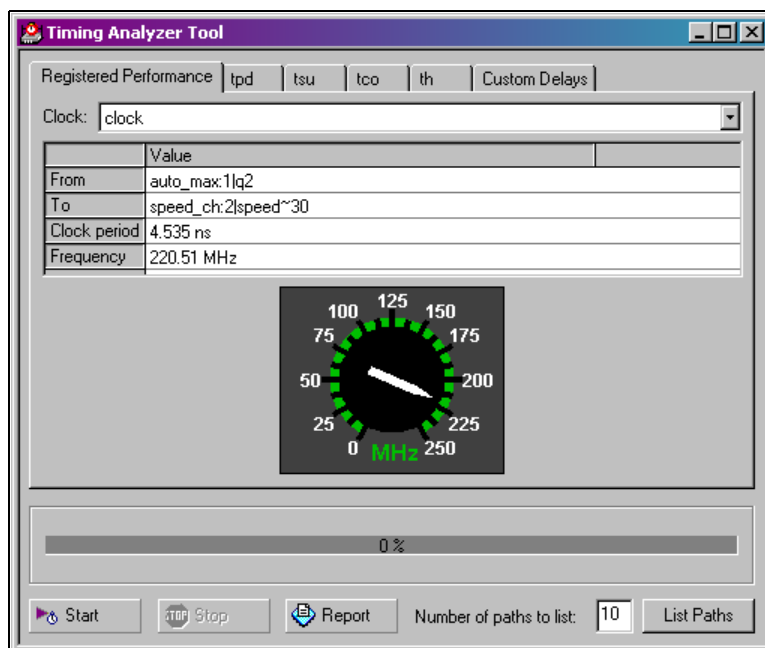
To further reduce compilation times, turn on **Limit to one fitting attempt** in the **Fitter Settings** page in the **Settings** dialog box (Assignments menu).

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 control 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 Analyzer to analyze more complex clocking schemes than is possible with the MAX+PLUS II Timing Analyzer.

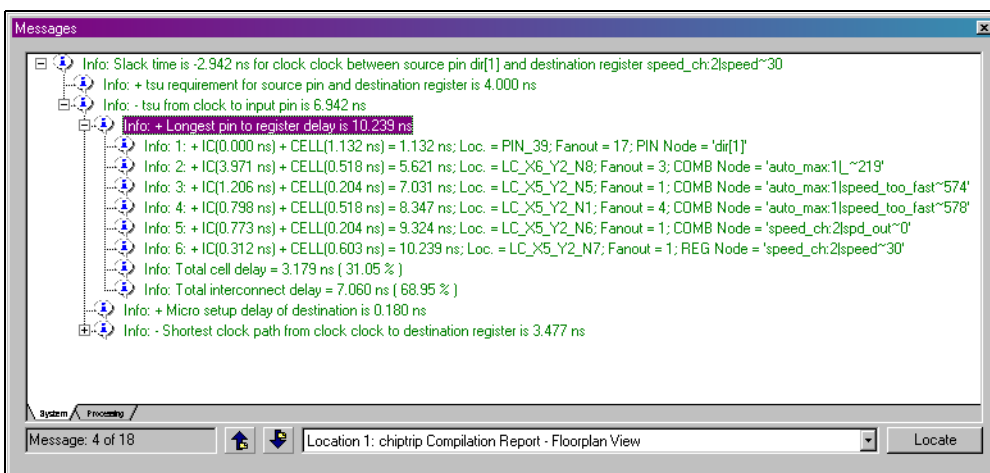
Launch the Timing Analyzer Tool by choosing **Timing Analyzer** (MAX+PLUS II menu) or by choosing **Timing Analyzer Tool** (Tools menu) (see [Figure 2–17](#)). To start the analysis, click **Start** in the Timing Analyzer Tool or choose **Start > Start Timing Analyzer** (Processing menu).

Figure 2–17. 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 crossing 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 2–18).

Figure 2–18. Messages Window Displaying Detailed Timing Information

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

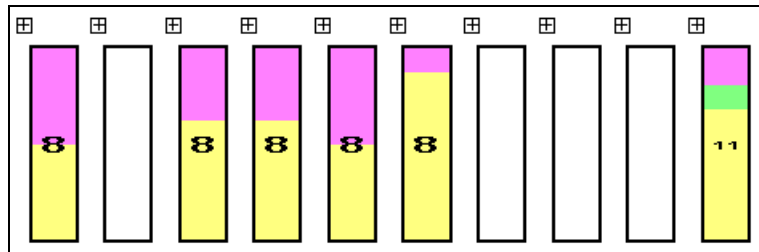
Timing Closure Floorplan

The Quartus II Timing Closure Floorplan is similar to the MAX+PLUS II Floorplan Editor but has many improvements to help you more effectively debug and view 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 between the Interior MegaLABs (where applicable), Interior LABs, and the Field view.

The Interior LABs view hides cell (logic cell, Adaptive Logic Module [ALM], and macrocells) details and shows LAB information (see [Figure 2–19](#)). You can display the number of cells used in each LAB by selecting **Show Usage Numbers** (View menu).

Figure 2–19. 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-click pop-up menu).



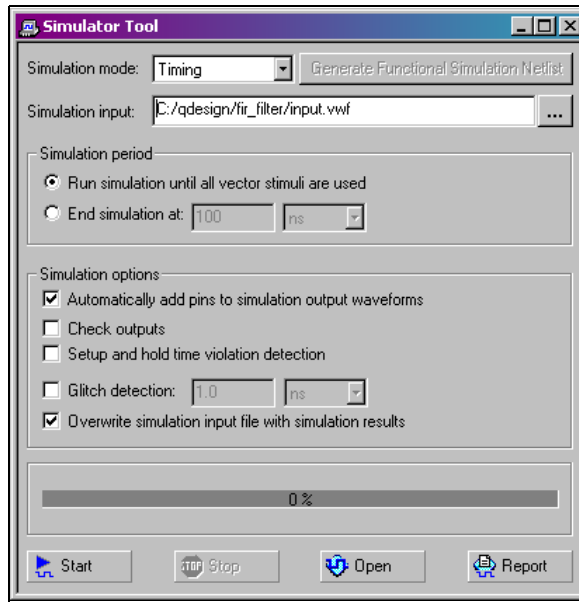
For more information, see the *Timing Closure Floorplan* chapter in Volume 2 of the *Quartus II Handbook*.

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 2–20). When performing timing simulation, the simulator uses place-and-route timing information.

Figure 2–20. Quartus II Simulator Tool

You can use Tcl commands, Vector Waveform Files, text-based Vector Files, 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.

You can also follow step-by-step instructions to help you set simulation settings. To start the Simulator Setting Wizard, choose **Simulator Settings Wizard** (Assignments menu).

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, a Standard Delay Format Output File, and a device-specific atom file.

Specify your EDA simulation tool by selecting the tool under **Tool name** on the **EDA Tool Settings > Simulation** page of the **Settings** dialog box (Assignment 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 Excel-based power calculator available on the Altera Web Site at www.altera.com, or in the Quartus II software.

You can use the Excel-based power calculator by entering device resource and performance information. Or, you can use the Quartus II software-generated power estimation file and import it into the power calculator. To generate the power estimation file, choose **Generate Power Estimation File** (Project menu). The power calculator spreadsheet supports the Stratix, Stratix GX, and Cyclone device families.

To report estimated power using the Quartus II software, simulate your design with an input stimulus file. You can use the Quartus II simulator or an EDA simulation tool to perform the simulation. The Cyclone, MAX 7000B, MAX 7000AE, MAX 3000A, Stratix II, Stratix GX, and Stratix device families are supported by this method for estimating power.

If you use the Quartus II simulator, enter the required information in the **Power Estimation** dialog box available from the **Simulator** page of the **Settings** dialog box (Assignments menu). Power estimation results appear in the simulator summary page of the simulation report after a timing simulation.



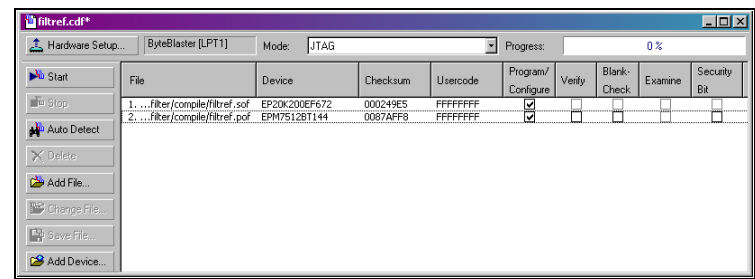
For more information on early power estimation, see the *Early Power Estimation* chapter in Volume 3 of the *Quartus II Handbook*. For more information about how to use the simulation-based power estimation feature in Quartus II, see the *Simulation-Based Power Estimation* chapter in the *Quartus II Handbook*.

Programming

The Quartus II Programmer has the same functionality as the MAX+PLUS II Programmer including programming, verifying, examining, and blank checking operations. To improve usability the Quartus II Programmer displays all programming-related information in one window (see [Figure 2–21](#)).

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

Figure 2–21. Programmer Window



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. To restore the programmer settings, browse to the CDF in the **Open** dialog box (File menu).

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 changes according to the window that is active (see [Figures 2-22](#) and [2-23](#)). In the following example, the Graphic Editor window is active.

Figure 2-22. MAX+PLUS II Quick Menu

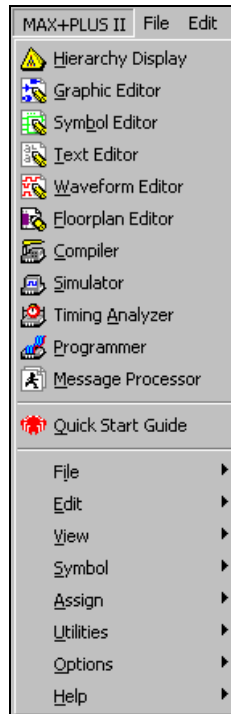
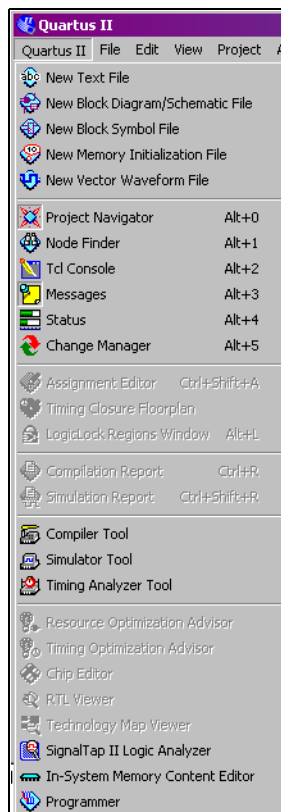


Figure 2–23. Quartus II Quick Menu

Quartus II Command Reference for MAX+PLUS II Users

NA means either Not Applicable or Not Available.

If the command is not listed, then the command is the same in both tools.

Table 2–3. Quartus II 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
 File > Project > Save & Compile (Ctrl+L)	 Processing > Start Compilation (Ctrl+L)
 File > Project > Save & Simulate (Ctrl+Shift+L)	 Processing > Start Simulation (Ctrl+I)

Table 2–3. Quartus II Reference for MAX+PLUS II Users (Part 2 of 11)










MAX+PLUS II Software	Quartus II Software
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
(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

Table 2–3. Quartus II Reference for MAX+PLUS II Users (Part 3 of 11)






MAX+PLUS II Software	Quartus II Software
(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
(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

Table 2–3. Quartus II Reference for MAX+PLUS II Users (Part 4 of 11)






















MAX+PLUS II Software	Quartus II Software
 (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+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)
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

Table 2–3. Quartus II Reference for MAX+PLUS II Users (Part 5 of 11)
















MAX+PLUS II Software	Quartus II Software
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
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

Table 2–3. Quartus II Reference for MAX+PLUS II Users (Part 6 of 11)







MAX+PLUS II Software	Quartus II Software
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
(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

Table 2–3. Quartus II Reference for MAX+PLUS II Users (Part 7 of 11)










MAX+PLUS II Software	Quartus II Software
(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
(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

Table 2–3. Quartus II Reference for MAX+PLUS II Users (Part 8 of 11)










MAX+PLUS II Software	Quartus II Software
(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)
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

Table 2–3. Quartus II Reference for MAX+PLUS II Users (Part 9 of 11)















MAX+PLUS II Software	Quartus II Software
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
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
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

Table 2–3. Quartus II Reference for MAX+PLUS II Users (Part 10 of 11)






MAX+PLUS II Software	Quartus II Software
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
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

Table 2–3. Quartus II Reference for MAX+PLUS II Users (Part 11 of 11)

MAX+PLUS II Software	Quartus II Software
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

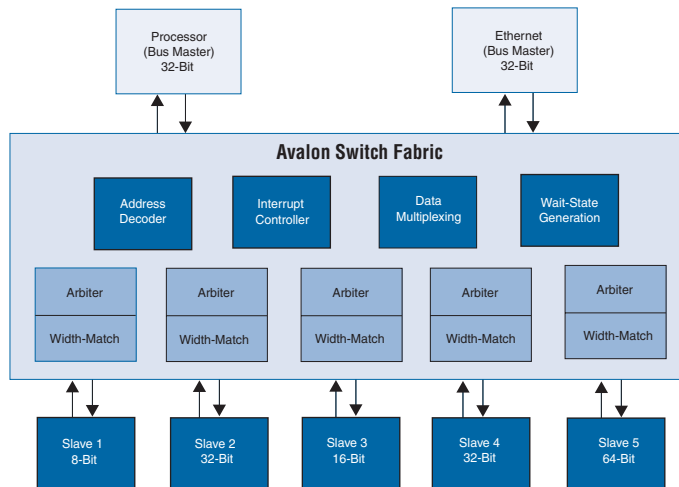
Introduction

The Altera® SOPC Builder system development tool provides a powerful platform for creating memory-mapped systems based on processors, peripherals, and memories that are internal or external to the FPGA. You can use SOPC Builder to define and implement a complete system in a fraction of the time required using traditional, manual system-on-a-chip (SoC) methods. SOPC Builder is included in Altera's Quartus® II software, giving Quartus II users immediate access to this development tool.

SOPC Builder automates the task of integrating the address-based read/write interfaces to hardware design modules. By integrating modules automatically, SOPC Builder dramatically simplifies the task of creating high-performance system-on-a-programmable-chip (SOPC) designs. In traditional SoC design, you must manually connect all of the system components. Using SOPC Builder, you need only specify the peripherals; SOPC Builder generates the interconnect logic automatically, including address decoding, data-path multiplexing, wait-state generation, interrupt controller, and data-width matching.

The outputs of SOPC Builder are hardware design language (HDL) files that define all components of the system, and a top-level HDL design file called the system module that ties all these components together.

[Figure 3-1 on page 3-2](#) shows an example of a multi-master system module connecting multiple master and slave peripherals. SOPC Builder generates the Avalon™ switch fabric that contains logic to manage the connectivity of all modules in the system.

Figure 3–1. Example of a System Module Generated by SOPC Builder

SOPC Builder Peripherals

Altera and other developers provide SOPC Builder components that range from simple blocks of fixed logic, to complex, parameterized, and dynamically-generated subsystems. Available SOPC Builder hardware components include:

- Microprocessors
- Microcontroller peripherals
- Timers
- Serial communication interfaces, such as UART and 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 (ASSPs)
 - ASICs

You can use the SOPC Builder to connect any block of logic that uses the Avalon interface or the AMBA™ advanced high-performance bus (AHB) interface. Most SOPC Builder peripherals use the Avalon interface.

Check the Altera web site at www.altera.com for up-to-date information about available SOPC Builder Ready components.

SOPC Builder Ready Functions

Altera awards the SOPC Builder Ready certification to intellectual property (IP) design functions 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 offers a free “test drive” of IP functions through the OpenCore® and OpenCore Plus evaluation features. You can verify the functionality quickly and easily of a IP function both in simulation and in hardware, as well as evaluate the size and performance before making the purchase decision.



You can download OpenCore and OpenCore Plus 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.

User-Defined Peripherals

SOPC Builder provides a simple method for you to develop and connect your own modules:

1. You create a block of logic with an Avalon or AHB interface in either Verilog HDL or VHDL.

With the Avalon interface, user-defined peripherals need to adhere only to a simple interface based on address, data, read-enable, and write-enable signals.

2. Use the Interface to User Logic Wizard to import your HDL files into SOPC Builder.

You use the wizard to map the input and output signal names to Avalon signal types, specify the timing requirements, and specify simulation files.

3. Instantiate the custom module in the same manner as for other SOPC Builder Ready components.

User-defined peripherals can be instantiated multiple times in a single design, and can be used in other system designs.

Embedded Software Applications

You can specify and integrate software components for microprocessor-based systems. For each processor included in the system, SOPC Builder generates the following applications software-specific information:

- Software components
- Header files (C and assembly)
- Generic C drivers
- Operating system (OS) kernels
- Middleware libraries

These files form a custom software development kit (SDK) for each processor. For more details, see [“SDK Option” on page 3–10](#).

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 control, data and address signals, and arbitration logic connecting master and slave peripherals. The Avalon switch fabric is implemented as a configurable architecture that matches the interface ports, logical behavior, and signal sequencing of the specific peripherals connected to the system.

The Avalon switch fabric is designed for both simplicity and performance. The Avalon signal types are straightforward, and you can design peripherals generically without knowing about the other peripherals connected to the system. The Avalon switch fabric implements a point-to-point connection between all master-slave pairs that require connection.

The multi-master Avalon switch fabric maximizes system throughput by allowing all master peripherals to transfer data in parallel. The Avalon switch fabric also supports pipelined transfers, so that master-slave pairs achieve the maximum throughput possible. SOPC Builder automatically implements slave-side arbitration whenever necessary. With slave-side arbitration, master transfers stall only when multiple master ports attempt to access the same slave port at the same time.

Automatic Generation

SOPC Builder generates the Avalon switch fabric to connect master and slave peripherals. Use the SOPC Builder to define the connectivity between peripherals. With this information, SOPC Builder automatically creates and connects the HDL modules.

Because of the SOPC Builder interface, your view of the Avalon switch fabric is usually limited to the specific ports that relate to the connection of custom Avalon peripherals. For SOPC Builder Ready IP functions, SOPC Builder automatically connects the Avalon ports correctly, making it unnecessary for you to consider the interfaces for these IP functions.

Function

The Avalon switch fabric provides the following services to connected peripherals:

- *Data-Path Multiplexing*—Multiplexers in the Avalon switch fabric transfer data from the selected slave peripheral to the appropriate master peripheral.
- *Address Decoding*—Address decoding logic produces chip-select signals for each peripheral. This simplifies peripheral design, because individual peripherals do not need to decode the address lines to generate chip-select signals.
- *Pipelined Transfer Capabilities*—Latency-aware peripherals are capable of initiating multiple read transfers in succession without waiting for the first transfer to complete, also known as pipelining data transfers. Transfers with latency allow master-slave pairs to achieve maximum throughput performance, even though the first transfer may require several cycles of latency to present valid data.
- *Wait-State Generation*—Wait-state generation extends transfers by one or more clock cycles, for the benefit of peripherals with special synchronization needs. Wait states can be generated to stall a master peripheral in cases when the target peripheral cannot respond in a single clock cycle. Wait states are also generated in cases when read-enable and write-enable signals have setup or hold time requirements.
- *Dynamic-Bus Sizing*—Dynamic-bus sizing hides the details of interfacing narrow Avalon slave ports to a wider Avalon master port, and vice versa. For example, in the case of a 32-bit master read transfer from a 16-bit memory, dynamic-bus sizing automatically executes two slave read transfers to fetch 32 bits of data from the 16-bit memory device. This reduces the logic and/or software complexity in master peripherals, because the master port does not need to be aware of the physical nature of the slave port.
- *Interrupt-Priority Assignment*—When one or more slave ports generate interrupts, the Avalon switch fabric passes the interrupts to appropriate master peripherals.

System Generation

After you add all peripherals and specify all necessary system parameters, SOPC Builder is ready to generate the system. During system generation, SOPC Builder creates the following items:

- The HDL files for the top-level system module and each component in the system
- A Block Symbol File (.bsf) representation of the top-level system module
- The SDK for application software development
- ModelSim® simulation project files
- A Tcl script that sets up all of the files needed for compilation in the Quartus II software

After you generate the system module, it can be compiled directly by the Quartus II software, or included in a larger design.

Simulation Model & Testbench

During system generation, SOPC Builder optionally can output a simulation environment that accelerates the system simulation effort. SOPC Builder generates both a simulation model and a testbench for the entire system. You can simulate your custom systems with minimal effort, immediately after generating the system with SOPC Builder. For more information, see the [“Simulation Option” on page 3–12](#).

Using SOPC Builder

Launch SOPC Builder in the Quartus II software as follows:

1. Open a project in the Quartus II software.
2. Choose **SOPC Builder** (Tools menu) in the Quartus II software to run SOPC Builder.

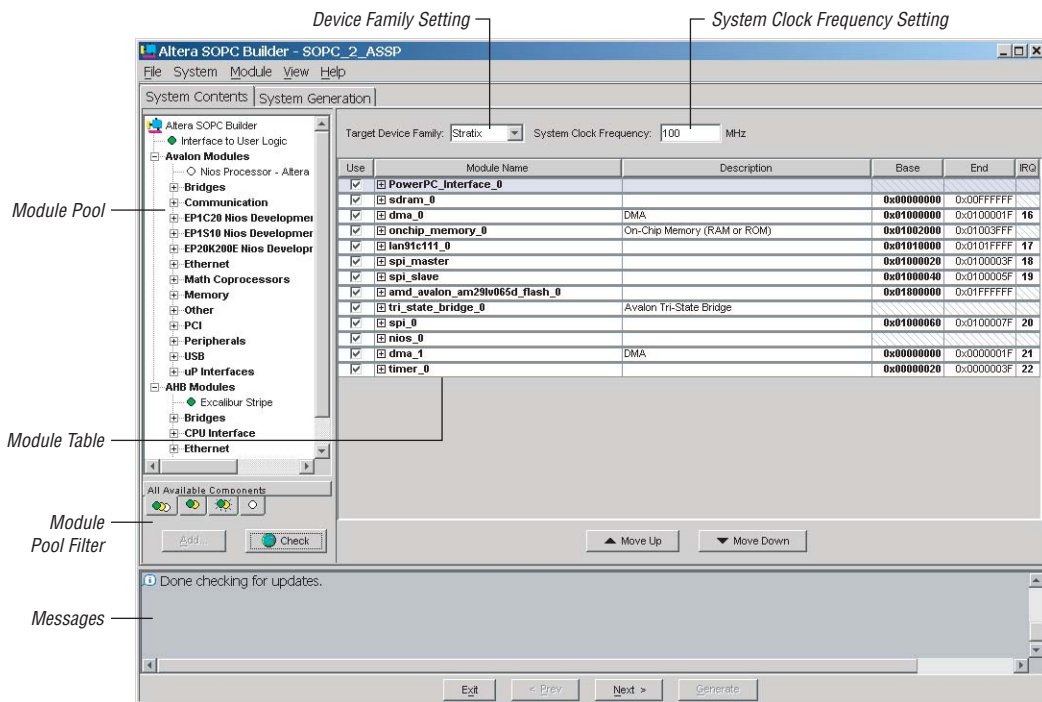
SOPC Builder provides an easy-to-use, table-oriented user interface for defining systems. SOPC Builder contains two primary pages, **System Contents** page and **System Generation** page.

In addition, certain peripherals may add system dependency pages that allow you to specify refinements and relationships with other peripherals in the system.

System Contents Page

You select components and define the system on the **System Components** page. The page is split into two sections: the module pool and the module table. The module pool lists all the available components, while the module table displays the components added to the current system. [Figure 3–2](#) shows an example of the **System Contents** page.

Figure 3–2. Systems Contents Page



Module Pool

The module pool shows the available library of components organized according to category. Each component appears with a colored dot next to its name. The colors of the dots have the following meanings:

- **Green dot**—Indicates fully licensed components.
- **Yellow dot**—Indicates that a component is available for system design evaluation in some limited form, typically subject to a hard time out or reduced functionality.
- **White dot**—Indicates SOPC Builder Ready components available from Altera or partner vendors that are not currently installed on the PC. Evaluation versions of these components can be downloaded from the web for free.

You can use the module pool to filter the display for available components, installed components, components available on the web, or components with updates available on the web. In addition, if you have an Internet connection, the module pool is updated to show new components from Altera and partner vendors as they become available.

Module Table

The module table lists components that are included in the current system. Additionally, the module table describes the following elements:

- Connectivity between master and slave ports
- Addresses for each slave port
- Interrupt controller (IRQ) assignments for each slave port
- Arbitration priorities for slave ports shared by multiple-master ports

If **Show Master Connections** (View menu) is turned on, the left side of the module table displays the connectivity between master ports and slave ports. You can selectively connect any master port to any slave port. Whenever two or more master ports share (i.e., have access to) the same slave port, SOPC Builder automatically inserts an arbiter to grant access to the slave when simultaneous requests occur.

To view arbitration priorities, choose **Show Arbitration Priorities** (View menu).

You can connect any master port to any slave port if 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.



For more information on master/slave connections and arbitration priorities, see *Application Note 184: Simultaneous Multi-Mastering with the Avalon Bus*.

Additional Settings

The **System Contents** page includes additional options as shown in [Table 3–1](#).

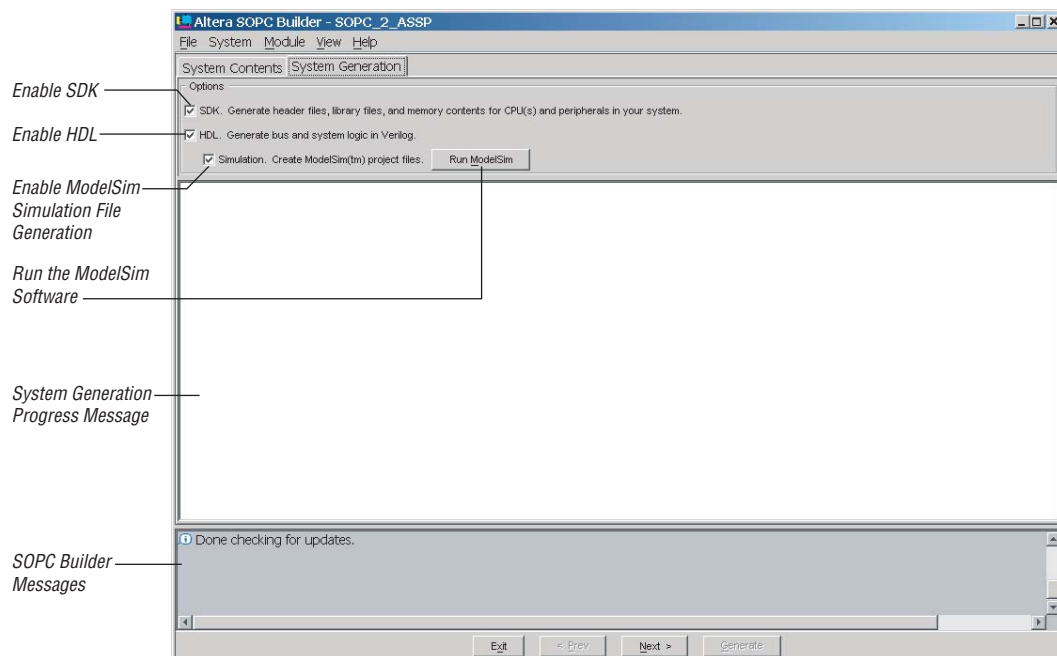
Table 3–1. System Contents Page Options	
Option	Description
Target	Select one of the listed Development boards. The Target Device Family and System Clock Frequency fields are automatically populated. Select the Unspecified Board to select the Target Device Family and to set the System Clock Frequency manually.
Target Device Family	Select the target FPGA device family from the Target Device Family list. SOPC Builder takes advantage of the architectural features of a specified device family when generating the system module. (1)
System Clock Frequency	The System Clock Frequency setting specifies the master clock frequency that drives the system module. Many peripherals use the system clock frequency to generate clock dividers, baud rate generators, etc. SOPC Builder's built-in testbench generator also uses the setting to simulate a clock of the requested frequency. (1)

Note to [Table 3–1](#):

- (1) The Quartus II software does not detect this setting automatically. The setting must also be specified in the Quartus II software.

System Generation Page

The **System Generation** page includes settings to control the generation process for the hardware design files, simulation model, and the SDK. [Figure 3–3 on page 3–10](#) shows the **System Generation** page.

Figure 3–3. System Generation Page *Note (1)***Note to Figure 3–3:**

- (1) Some SOPC Builder Ready processor components may modify the look of this page to allow the user better access to the processors software tool chain. Please refer to the processors documentation for more information.

SDK Option

When the SDK option is turned on, SOPC Builder creates a custom SDK for each CPU in the system. This SDK contains software files (drivers, libraries, and utilities) for any system components that provide software-support files.

You can build software applications for Excalibur™ devices as part of the generation process. The 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.

You should save any file you have edited with a unique filename to prevent the file from being overwritten in a subsequent system generation. Altera recommends that your source code be stored in one of the following locations:

- **src** directory
- A subdirectory of the **src** directory
- A directory on the same level as the **src** directory in the SDK

You should provide all the SDK files to the software engineers developing application code for the system every time you generate or update the system hardware module.



Certain components, such as the Nios® II embedded processor, may modify the contents on the page to provide better access to the software development environment for the processor. Please refer to the CPU component documentation for more information.

HDL Option

To generate HDL files that describe the system, turn on the **HDL** option. The files are Verilog HDL or VHDL, depending on which language you specify when starting SOPC Builder. The Verilog HDL files contain the following components:

- An instance of every component in the system
- The Avalon switch fabric tailored to connect all components in the system. See [“Avalon Switch Fabric” on page 3–4](#) for more details.
- A simulation model and a simulation testbench, depending on the **Simulation** option. See the [“Simulation Option”](#) section for more details.

Simulation Option

To generate a simulation model and a test bench to simulate a custom model, turn on the **Simulation** option. 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 (e.g., memory models)

There are custom simulation files associated with each peripheral. SOPC Builder copies these files into a simulation directory during system generation. The simulation directory is separate from the directory for synthesizable HDL. The system provides files for each peripheral that address the needs of both environments.

Simulating with ModelSim

SOPC Builder generates a ModelSim project directory that includes the following files:

- 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 displays an initial set of bus interface waveforms
- A ModelSim Project File (**.mpf**) for the current system

Run the ModelSim software in SOPC Builder by clicking **Run ModelSim**. If the ModelSim software is not in your search path, specify the location in the **SOPC Builder Setup** dialog box (File menu).

Simulating with Other Simulators

You can use the SOPC Builder-generated simulation files with simulation software other than ModelSim. However, you cannot use the files generated for ModelSim (**.tcl**, **.do**, **.mpf**, etc.) directly. You can inspect these files and use them as a basis for setting up similar capabilities in other simulation tools.

System Dependency Pages

Certain components, such as a CPU like the Nios II embedded processor, display an additional page called a system dependency page in SOPC Builder. System dependency pages display additional parameters or associations to be specified for a component. For example, you can

specify the relationship between a CPU and memory components to indicate which memory is to be used as the program memory and which is to be used as data memory. For components that use system dependency pages, a separate system dependency page is displayed for each instance of the component added to a system.

Generating a System

After you have specified the component and selected the generation options, click the **Generate** button to cause SOPC Builder to generate the system. This button is available from any page in the SOPC Builder user interface.

As the generation process proceeds, SOPC Builder displays messages and information in the system generation progress messages box. SOPC Builder displays the message “Generation Complete” when it is finished, and places a log file in the root directory of the project.

Further Information

For more information on using SOPC Builder, see the following documents:

- *Avalon Specification*—The Avalon interface specification is useful for developers creating custom peripherals. It defines terms and concepts of SOPC designs based on the Avalon interface used for connecting on-chip processors and peripherals into a system-on-a-programmable chip. Avalon interface signal functions and timing are defined.
- *Application Note 184: Simultaneous Multi-Mastering with the Avalon Bus*—This application note describes the simultaneous multi-master Avalon switch fabric with an explanation about how it differs from traditional arbitration schemes. It includes an in-depth explanation of bus arbitration priorities and most commonly used configurations for Nios embedded system design.
- *Application Note 333: Developing Peripherals for SOPC Builder*—This application note describes the process for developing Avalon peripherals that can be integrated into SOPC Builder-generated systems. Topics include how to define an Avalon interface for a custom peripheral, how to import HDL files into SOPC Builder, and how to provide software drivers for the peripheral. Example designs are provided for hands-on experience importing user designs into SOPC Builder.

- *Application Note 320: OpenCore Plus Evaluation of Megafunctions*—Altera MegaCore® functions and AMPP megafunctions are reusable blocks of intellectual property that you can customize and use in a design. With Altera's free OpenCore® Plus evaluation feature you can simulate the behavior of a megafunction, verify functionality, and generate a time-limited device to verify your design.
- *Application Note 323: Using SignalTap II Logic Analyzer for Embedded Designs using SOPC Builder*—SignalTap® II is a system-level debugging tool that captures and displays real-time signals in a SOPC design. By using a SignalTap II Embedded Logic Analyzer (ELA) in SOPC Builder generated systems, you can observe the behavior of peripherals in response to software execution.
- *Simultaneous Multi-Mastering with the Nios Processor Tutorial*—This tutorial describes how to optimize an embedded system's performance using the simultaneous multi-master bus architecture. It describes the features in SOPC Builder that easily allow the customization of a system architecture to define a new architecture to improve the example design's performance.
- *Using SOPC Builder with Excalibur Devices Tutorial*—This tutorial introduces the process of using SOPC Builder to generate systems with Excalibur devices. It shows how to use SOPC Builder and the Quartus II software to create and process Excalibur device system modules to interfaces with components provided on the Excalibur development board.
- *SOPC Builder PTF File Reference Manual*—This reference manual is for IP developers creating library components for SOPC Builder. This manual contains reference material on the internal workings of the peripheral template file (.ptf) structure and the generation phases of SOPC Builder. This manual is recommended for advanced system designers with basic familiarity of SOPC Builder.

Introduction

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 migrating to the corresponding HardCopy device for high-volume production.

HardCopy structured ASICs provide the following key benefits:

- Improves performance, on the average, 50% over the corresponding FPGA
- Lowers power consumption, on the average, 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 less than 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 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

Features

The Quartus II software version 4.1 contains several powerful features that facilitate design of HardCopy devices:

- **HARDCOPY_FPGA_PROTOTYPE Devices**
These are Stratix FPGA devices with features identical to HardCopy Stratix devices. You can use these FPGA devices to prototype your designs and evaluate the functionality in silicon.
- **HardCopy Timing Optimization Wizard**
Using this feature, you can target your designs 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 performance and power consumption.
- **Placement Constraints**
Location and LogicLock® constraints are supported at the HardCopy floorplan level to improve overall performance.
- **Improved Timing Estimation**
The Quartus II software version 4.1 determines routing and associated buffer insertion for the design, and provides the Timing Analyzer with more accurate information on the delays than was possible in previous versions of the Quartus II software. The buffer insertion information is exported in the Quartus Archive (.qar) file, so the back-end design team can insert the buffers correctly.
- **Design Assistant**
This feature checks your design for compliance with HardCopy 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 with a "single click."
- **HardCopy Stratix Power Calculator**
This calculator is launched from the Quartus II software to estimate power consumed by the HardCopy Stratix devices.

HARDCOPY_FPGA_PROTOTYPE, HardCopy Stratix, and Stratix Devices

You can use the HARDCOPY_FPGA_PROTOTYPE devices available in Quartus II software to quickly target your designs to the actual resources and package options available in the equivalent post-migration HardCopy Stratix device. You can also 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 high-volume production device. It also minimizes risk, assures samples in about eight weeks, and guarantees first-silicon success.

Table 4-1 compares HARDCOPY_FPGA_PROTOTYPE devices, Stratix devices, and HardCopy Stratix devices.

Table 4-1. Qualitative Comparison of HARDCOPY_FPGA_PROTOTYPE with Stratix and HardCopy Stratix Devices		
Stratix Device	HARDCOPY_FPGA_PROTOTYPE Device	HardCopy Stratix Device
FPGA	Virtual FPGA	Structured ASIC
(Reference) (1)	Architecture identical to Stratix FPGA	Architecture identical to Stratix FPGA
(Reference) (1)	Resources identical to HardCopy Stratix device	M-RAM resources different than Stratix FPGA in some devices
Ordered through Altera Part Number	Cannot be ordered	Ordered by Altera part number

Note to Table 4-1:

- (1) Reference indicates that the HARDCOPY_FPGA_PROTOTYPE and HardCopy Stratix devices are compared with the Stratix device for this attribute.

Table 4-2 lists the resources available in each of the HardCopy Stratix devices.

Table 4-2. HardCopy Stratix Device Physical Resources (Part 1 of 2)								
Device	LEs	Approx. ASIC Equivalent gates (K) (1)	M512 Blocks	M4K Blocks	M-RAM Blocks	DSP Blocks	PLLs	Max. User I/Os
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

Table 4–2. HardCopy Stratix Device Physical Resources (Part 2 of 2)

Device	LEs	Approx. ASIC Equivalent gates (K)(1)	M512 Blocks	M4K Blocks	M-RAM Blocks	DSP Blocks	PLLs	Max. User I/Os
HC1S60F1020	57,120	570	574	292	6	18	12	773
HC1S80F1020	79,040	800	767	364	6 (2)	22	12	773

Note to Table 4–2:

- (1) Does not include DSP blocks, on-chip RAM, or PLLs.
- (2) The M-RAM resources for these HardCopy devices differ from the corresponding Stratix FPGA.

For a given density, 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.

The `HARDCOPY_FPGA_PROTOTYPE` device aids in designing the structured ASIC, and provides a path to having an FPGA-proven design prior to migration. The physical entity for the `HARDCOPY_FPGA_PROTOTYPE` device is the equivalent Stratix FPGA. Designs targeting `HARDCOPY_FPGA_PROTOTYPE` in the Quartus II software configure the equivalent Stratix FPGAs with HardCopy-structured ASIC resources. Therefore, it is normal to find unused resources in the Stratix FPGAs. The three types of 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.



For more information on HardCopy Stratix devices, see the *HardCopy Stratix Device Family Data Sheet* section of the *HardCopy Device Handbook*.



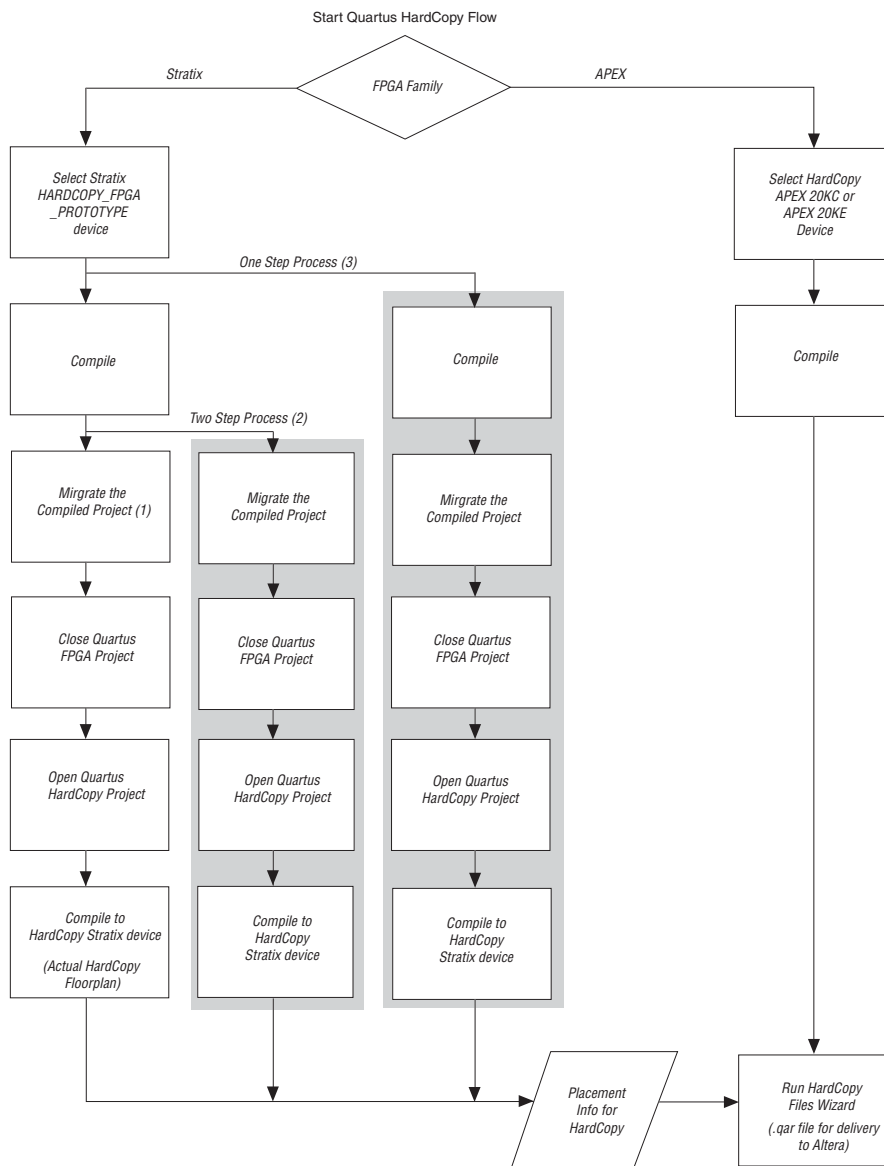
`HARDCOPY_FPGA_PROTOTYPE` devices are not available for the APEX 20K family.

HardCopy Design Flow



Figure 4–1 shows a HardCopy design flow diagram. The design steps are explained in detail in the following sections of this chapter.

For a detailed description of the HardCopy Timing Optimization wizard and HardCopy Files wizard, see “[HardCopy Timing Optimization Wizard Summary Page](#)” on page 4–9 and “[Generating the HardCopy Design Database](#)” on page 4–16.

Figure 4–1. HardCopy Design Flow Diagram**Notes for Figure 4–1:**

- (1) Migrate Only: 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

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 Devices

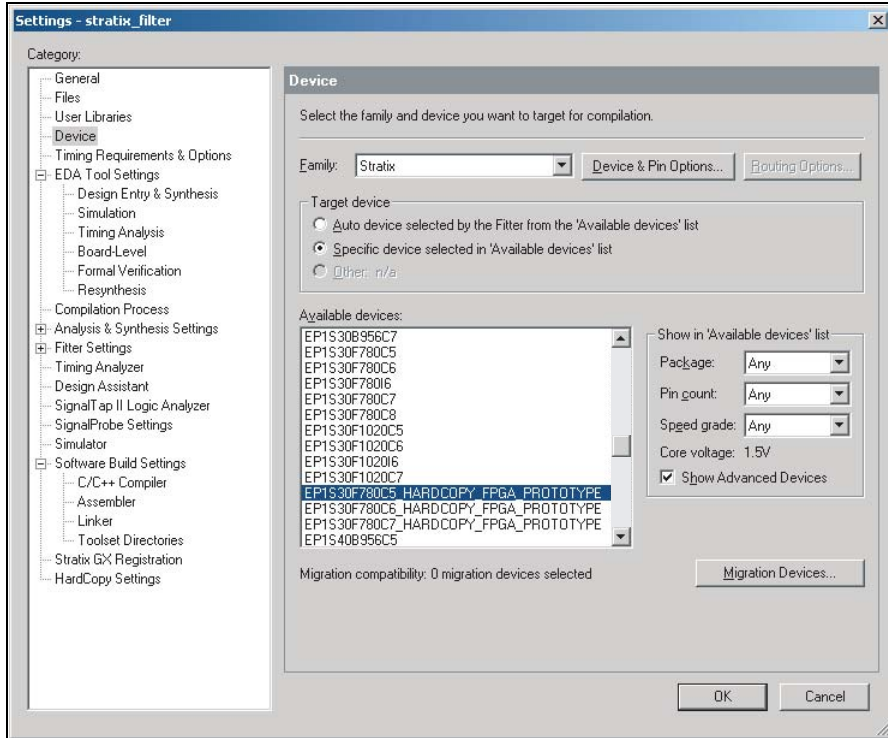
Targeting Designs to `HARDCOPY_FPGA_PROTOTYPE` Devices

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. You should also select the package, pin count, and speed grade, as shown in [Figure 4-2](#).

By choosing the `HARDCOPY_FPGA_PROTOTYPE` device, all the information necessary for the post-migration HardCopy Stratix device is included. The resources, package option, pin assignments, and all other data is produced. The netlist resulting from the compilation contains information about the electrical connectivity, resources used, I/O placements, and the resources unused in the FPGA device.

Figure 4–2. Selecting a `HARDCOPY_FPGA_PROTOTYPE` Device



3. Choose **Settings** (Assignments menu). In the **Category** list select **HardCopy Settings** and specify the external default clock jitter.

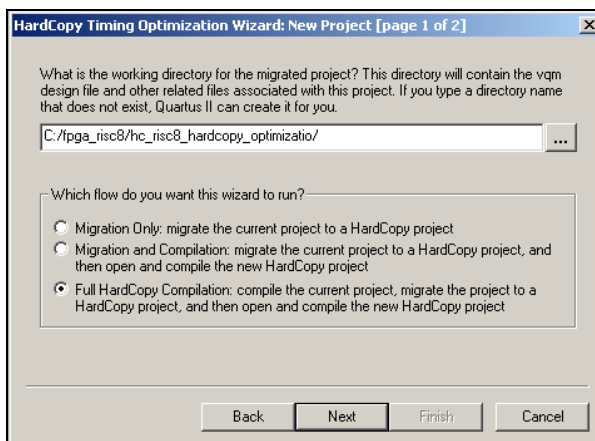
At this point, you are presented with the following three choices to target the designs to HardCopy Stratix devices, as shown in [Figure 4–3](#).

- **Migration Only:** You can select this option after compiling the project to migrate the project to a HardCopy project.

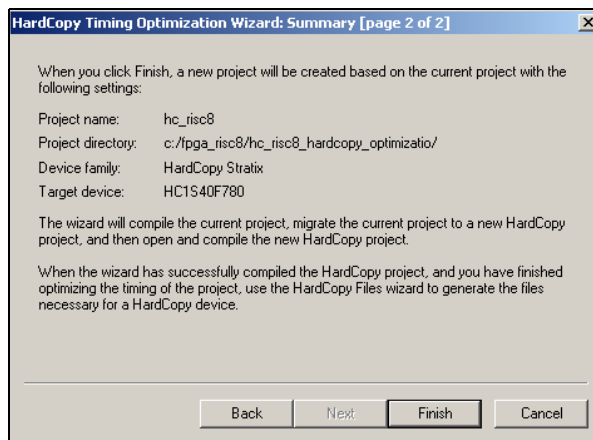
You can now perform the following tasks manually to target the design to a HardCopy Stratix device. See [“Performance Estimation” on page 4–10](#) if you need more information on how to perform these tasks.

- a. Close the existing project.
 - b. Open the migrated HardCopy project.
 - c. Compile the HardCopy project for a HardCopy Stratix device.
- **Migration and Compilation:** You can select this option after compiling the project and it results in the following actions:
 - Migrating the project to a HardCopy project
 - Opening the migrated HardCopy 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 project and compiling it for a HardCopy Stratix device

Figure 4–3. HardCopy Timing Optimization Wizard Options



After you make your selection, the HardCopy Timing Optimization wizard Summary page shows you details about the settings you made in the wizard, as shown in [Figure 4–4](#).

Figure 4–4. HardCopy Timing Optimization Wizard Summary Page

When either of the first two options in [Figure 4–3](#) are selected (**Migration Only** or **Migration and Compilation**), designs are targeted to HardCopy Stratix devices and optimized using the HardCopy placement and timing analysis to estimate performance. For details on the steps performed by the Quartus II software during **Full HardCopy Compilation and Migration and Compilation**, see [“Performance Estimation” on page 4–10](#). 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 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 is a powerful feature that helps you migrate designs from Stratix `HARDCOPY_FPGA_PROTOTYPE` devices to HardCopy Stratix devices.

HardCopy Floorplans & Timing Models

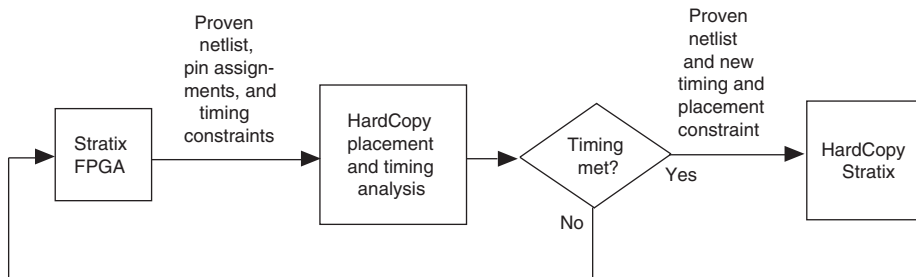
The Quartus II software version 4.0 and later supports HardCopy Stratix floorplans and HardCopy Stratix timing models. These features enable you to estimate the design's performance and power consumption in the migrated device. They reflect the actual placement of the design in the HardCopy Stratix device, and can be used to see the available resources, and the location of the resources in the actual device.

Performance Estimation

Figure 4-5 illustrates the design flow for estimating performance and optimizing your design. You can target your designs to `HARDCOPY_FPGA_PROTOTYPE` devices, and pass the design information to placement and timing analysis to estimate performance. In the event that the required performance is not met, you can modify your RTL source, optimize the FPGA design and estimate timing iteratively.



On average, HardCopy Stratix devices are 50% faster than their equivalent Stratix device. These performance numbers are highly design dependent, and final performance numbers must be obtained from Altera.

Figure 4–5. 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.

On completion of the HardCopy Timing Optimization wizard, the destination directory contains the Quartus II project file, and all files required for HardCopy 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 re-select the HardCopy Stratix devices 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.

4. Open the migrated Quartus II project created in the previous step.
5. Perform a full compilation.

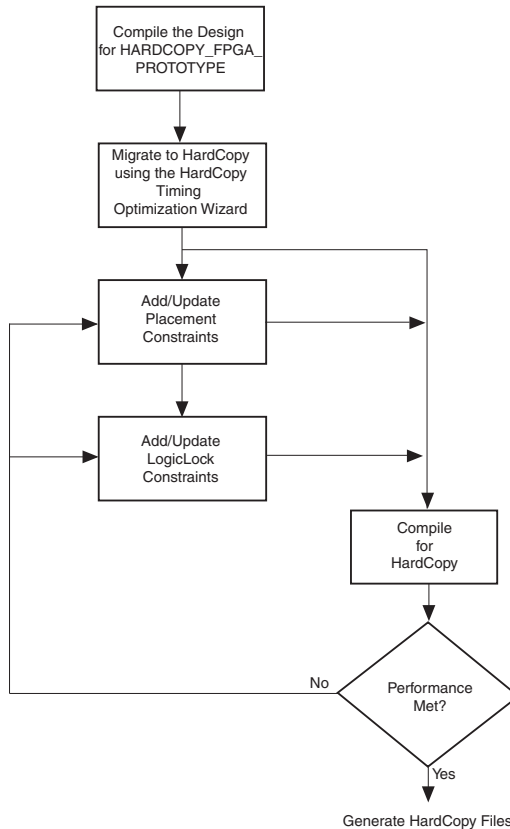
After successful compilation, the Timing Analysis section of the Compilation Report shows the performance of the design.



Performance estimation is not supported for HardCopy APEX 20K devices in the Quartus II software. Your design can be optimized by modifying the RTL code or the FPGA design and the constraints. You can discuss with Altera any performance improvement required with the HardCopy APEX 20K device.

Placement Constraints

The Quartus II software version 4.0 and later supports placement constraints and LogicLock regions for HardCopy Stratix devices. [Figure 4–6](#) shows an iterative process to modify the placement constraints until the best placement for the HardCopy Stratix device is achieved.

Figure 4–6. Placement Constraints Flow for HardCopy Stratix Devices

Location Constraints

LAB Assignments

The Quartus II software supports location constraints for HardCopy Stratix devices. You can make LAB-level assignments after migrating the `HARDCOPY_FPGA_PROTOTYPE` project, and before compiling the design for a HardCopy Stratix device, to achieve better performance. One important consideration for LAB reassignments is that the entire contents of a LAB is moved to another empty LAB. 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.

LogicLock Assignments

The LogicLock feature of the Quartus II software provides a block-based design approach. Using this technique you can create and implement each logic module independently, and integrate all optimized modules 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 “[An Example of Supported LogicLock Constraints](#)” on page 4–14. This modification is necessary because the floorplan of a HardCopy Stratix device is different from that of the Stratix device. If this modification did not occur, LogicLock assignments would lead to no-fit bad placements. Making the regions auto-size and floating maintains your module or entity LogicLock assignments, allowing you to easily adjust the LogicLock regions as required.

An Example of Supported LogicLock Constraints

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
```

Targeting Designs to HardCopy APEX 20KC and HardCopy APEX 20KE Devices

The Quartus II software version 4.0 and later also supports targeting designs to HardCopy APEX 20KC and HardCopy APEX 20KE device families. After compiling your design for one of the APEX 20KC or APEX 20KE devices supported by a HardCopy APEX device, run the HardCopy Files wizard to generate the necessary set of files for HardCopy migration.

Checking Designs for HardCopy Design Guidelines

When you develop a design with HardCopy migration in mind, follow design practices that ensure a straightforward migration process. 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. 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

The Design Assistant is run on a design in multiple ways. You must have run Analysis and Synthesis on the design before running the Design Assistant. 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.

Running Design Assistant

To run Design Assistant choose **Start > Start Design Assistant** (Processing menu).

The Design Assistant runs automatically when the HardCopy Timing Optimization wizard is launched. 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.

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. Reports include the settings, run summary, results summary, and details of the results and messages. The Detailed Results 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 *Design Guidelines for HardCopy Migration* chapter of the *HardCopy Device 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_directory>\<project_name>.hps`.

You can generate the archive of the HardCopy design database only after compiling the design to a HardCopy Stratix device. The Quartus II Archive file (**.qar**) is generated at the same directory level as the targeted project, either before or after optimization. The following are some of the files that are part of the archived files.

- `<project_name>.vqm`—This is a Verilog Quartus Mapping file.
- `<project_name>_pt_hcpy_v.tcl`—This is a set of PrimeTime scripts.
- `<project_name>_hcpy_v.sdo`—This is a standard delay file output (SDF) file.
- `<project_name>.sof`—This is a bitstream file used for programming the FPGA.
- `<project_name>.qsf`—This is the Quartus II assignments and settings file.

In addition to the archived file, the HardCopy Files wizard also creates a **hardcopy** directory that contains some of the important files that you can review to ensure the correctness of the design database.



The Design Assistant is run automatically 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 will 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 industry standard STA tool, PrimeTime. The following files, necessary for timing analysis with the PrimeTime tool, are generated by the HardCopy Files wizard:

- `<settings name>_hcpy.vo`—Verilog output format
- `<settings name>_hcpy_v.sdo`—standard delay file output (SDF) file
- `<settings 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 Stratix libraries 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 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 Calculator

The HardCopy Stratix power calculator 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 *Simulation-Based Power Estimation* chapter in Volume 3 of the *Quartus II Handbook*.

Opening HardCopy Stratix Power Calculator

The HardCopy Stratix power calculator 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 calculate the estimated power consumed under various conditions.

The calculator can also be opened independently of the Quartus II software by clicking the **HardCopy Stratix Power Calculator** link on the **HardCopy Design Utilities** web page on the Altera web site.



You must enter design-specific information manually if you run the calculator 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**.

The Quartus II software exports all the necessary data and displays the Power Calculator, a section of which is shown in [Figure 4–7](#).

Figure 4–7. HardCopy Stratix Power Calculator

Hardcopy Stratix Power Calculator - Summary

Enter Logic Array Information

- o Clock tree
 - Global Clock Network
 - Global Clock Region
 - Global Clock Fast
- o Logic element (LE)
- o Digital Signal Processing (DSP) Blocks
- o Phase-locked loops (PLL)
 - Enhanced Phase-locked loops
 - Fast Phase-locked loops

Enter RAM blocks, High-speed differential interface

- o RAM blocks
 - M512
 - M4K
 - M-RAM
- o High-speed differential interface (HSDI)
 - Receiver HSDI
 - Transmitter HSDI

Enter General I/O Information and Terminator Technology

- o General I/O power consumption
- o Terminator Technology

Enter Thermal Analysis Information

- o Total power
- o Thermal analysis
 - Without heat sink
 - With heat sink

Table 1. Device

Device	Package	Temperature Grade	V _{CCINT} (V)	P _{INT} (mW)	P _{IO} (mW)	P _{TOTAL} (mW)
HC1S25	672 FBGA	C-commercial	1.5	135.00	0.00	135.00

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 Calculator is run from the Quartus II software when the target is still `HARDCOPY_FPGA_PROTOTYPE` device. However, power is calculated for the HardCopy Stratix device, not for the FPGA.

Use the Stratix FPGA power calculator 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 20K Power Calculator

The HardCopy APEX 20K power calculator is also a web-based calculator that can be run from the Design Utilities section in the HardCopy APEX 20K web pages. You cannot open this calculator in the Quartus II software.

With the HardCopy APEX 20K power calculator, you can estimate the power consumed by HardCopy APEX 20K devices and design systems with the appropriate power budget.



HardCopy APEX 20K devices are generally expected to consume about 40% less power than the equivalent APEX 20K FPGAs.

Power Calculators for FPGAs

Stratix, Stratix GX, and Cyclone devices have Excel-based power calculators that are used to estimate the power consumed by the respective FPGAs. For access to these power calculators, see the respective Design Utilities web pages.

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*.

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. The same intellectual property cores and tools, including the SOPC Builder and DSP Builder, are used for HardCopy Stratix, HardCopy APEX 20KC, or HardCopy APEX 20KE devices.

Related Documents

For more information, refer to the following documentation:

- *Design Guidelines for HardCopy Migration* chapter of the *HardCopy Device Handbook*
- *Timing Closure in HardCopy Devices* chapter in Volume 2 of the *Quartus II Handbook*

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

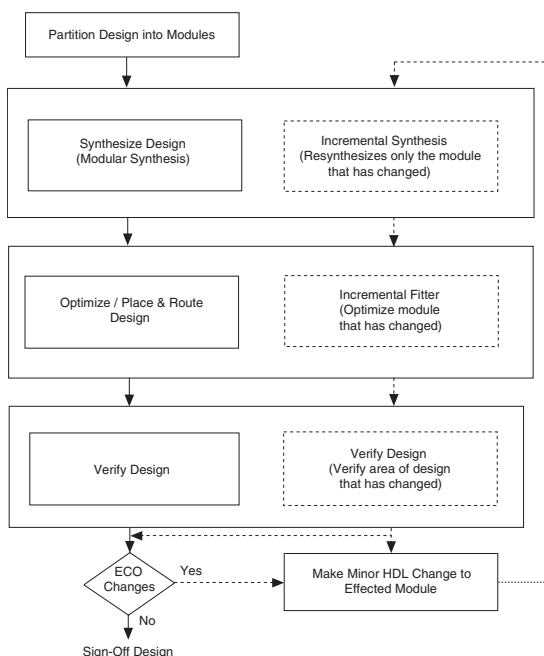
To leverage the previous design optimizations and to reduce the compilation time, there must be a means of performing an incremental compilation on the nodes with the new functionality and preserving the previous optimizations on the nodes that have not changed. Thus, a means of identifying nodes that 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 5–1 details the recommended design flow to support ECO changes at the HDL level.

Figure 5–1. 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.

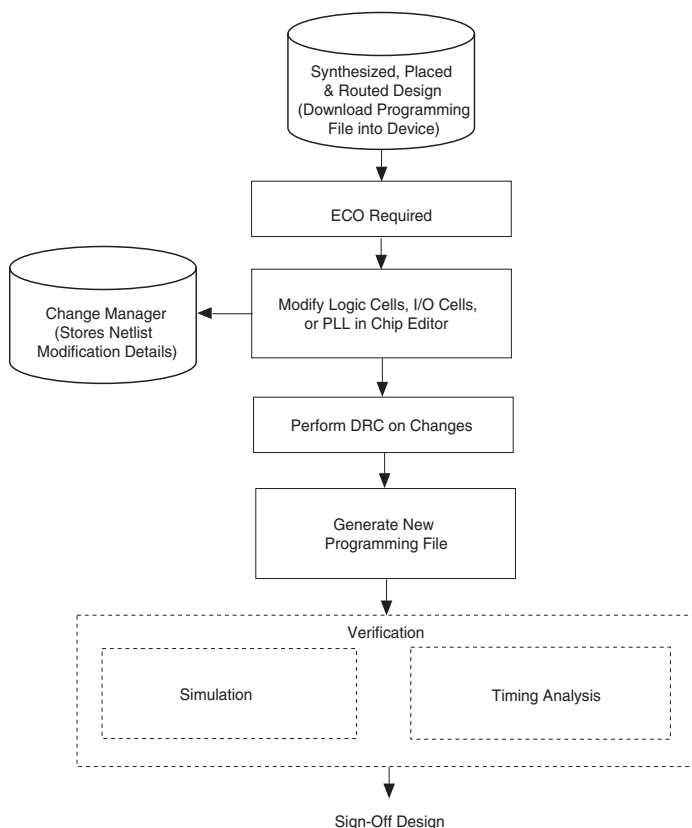
If this quick fix works, you do not need to change the HDL source code and rerun place-and-route. You should have the option to:

- 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 5–2 shows the flow for ECO changes at the netlist level.

Figure 5–2. 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.

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 6, Design Recommendations for Altera Devices
- Chapter 7, Recommended HDL Coding Styles

Revision History

The table below shows the revision history for Chapters 6 and 7.

Chapter(s)	Date / Version	Changes Made
6	June 2004 v.2.0	<ul style="list-style-type: none"> • Updates to tables, figures, and coding examples. • New functionality for Quartus 4.1.
	Feb. 2004 v1.0	Initial release.
7	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 4.1.
	Feb. 2004 v1.0	Initial release.

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 both prototyping and production. For optimal performance and better 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, and take advantage of the architectural features in the targeted device.



For specific HDL coding examples and recommendations, refer to the *Recommended HDL Coding Styles* chapter in Volume 1 of the *Quartus II Handbook*.

Synchronous FPGA Design Practices

The first step in a good design methodology is to understand the implications of your design practices and techniques. This section outlines some of the benefits of optimal synchronous design practices and the hazards involved in other techniques. Good synchronous design practices can help you consistently meet your design goals. Inherent problems with other design techniques can include reliance on propagation delays in a device, incomplete timing analysis, and possible glitches.

The basic principle of synchronous design is that 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.



For information about migrating designs to HardCopy devices, see the *Design Guidelines for HardCopy Migration* chapter in the *HardCopy Handbook*.

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

In Altera devices, 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.



Note that in order to meet setup and hold times 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 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.

Hazards of Asynchronous Design

In the past, designers have often used asynchronous techniques such as ripple counters or pulse generators in 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 large quantities of high-performance logic gates, registers, and memory, resource and performance trade-offs have changed. Now it is much 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 ordering 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 [“Recommended Design Techniques” on page 6–3](#). 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—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.

Recommended Design Techniques

When designing with hardware description language (HDL) code, it is important to understand how a synthesis tool interprets different HDL coding styles and what results to expect. Your coding style can affect logic utilization and timing performance. This section discusses some basic design techniques that ensure optimal synthesis results for designs targeted to Altera's devices while avoiding several common causes of

unreliability and instability. It is important to design your combinational logic carefully to avoid potential problems, and pay attention to your clocking schemes to maintain synchronous functionality.

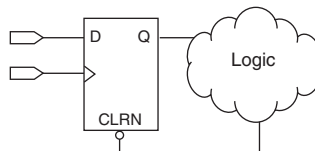
Combinational Logic Structures

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 (logic elements or adaptive logic modules). 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. In a synchronous design, all feedback loops should include registers. Combinational loops 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 6-1](#).

Figure 6-1. Combinational Loop through Asynchronous Control Pin



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.

Delay Chains

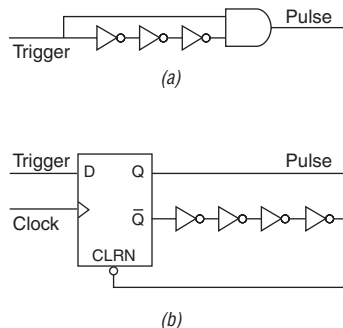
Delay chains occur when two or more consecutive nodes with a single fan-in and a single fan-out are used to cause delay. Often inverters are chained together to add delay. Delay chains are sometimes used to resolve race conditions created by other asynchronous design practices. As discussed above, delays in PLD designs can change with each place-and-route cycle. See [“Hazards of Asynchronous Design” on page 6–2](#) 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

Some designs use delay chains 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 6–2](#). These techniques are purely asynchronous and should be avoided.

Figure 6–2. Asynchronous Pulse Generators



In [Figure 6–2](#), part (a), 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.

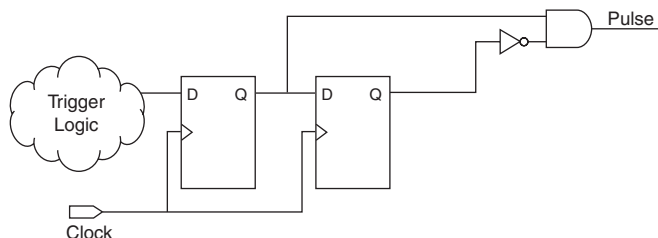
In [Figure 6–2](#), part (b), a register's output drives the same register's asynchronous reset signal through a delay chain. The register resets itself asynchronously after a certain delay.

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. It is difficult to reliably determine the width of the pulse when creating HDL code, and it cannot be set by electronic design automation (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. Multivibrators cause even more problems than pulse generators 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, it should be implemented using purely synchronous techniques, as shown in [Figure 6–3](#).

Figure 6–3. 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.

Latches

In digital logic, a latch 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. In some device architectures, latches add less delay and can be implemented using less silicon area than registers. However, FPGA architectures are based on registers. In FPGA devices, latches actually use more logic resources and lead to lower performance 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. When analyzing a design with a D latch, for example, 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.

Clocking Schemes

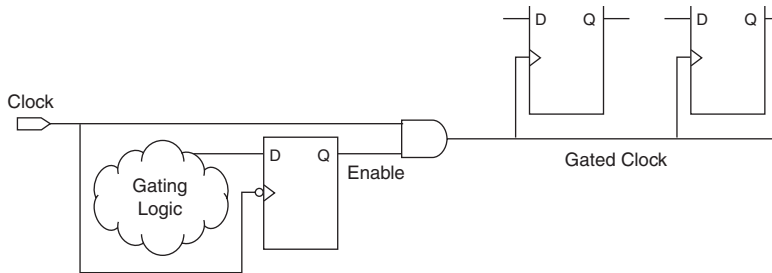
Like combinational logic, clocking schemes have a large effect on your design's performance and reliability. Avoid using internally generated clocks where possible because they can cause functional and timing problems in the design. Clocks generated with combinational logic can introduce glitches that 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.

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 6-4](#).

Figure 6-4. 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.

The combinational logic used to generate an internal clock also adds delays on the clock line. In some cases, logic 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 will be violated and the design will not function correctly. To reduce the clock skew within the clock domain, assign the generated clock signal to one of the global high-fan-out and low-skew clock networks in the FPGA device (if available). The Quartus® II software will automatically use global routing for high-fan-out control signals. 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.

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 6–7. To avoid glitches, you should not decode the outputs of a counter or a state machine to generate clock signals.

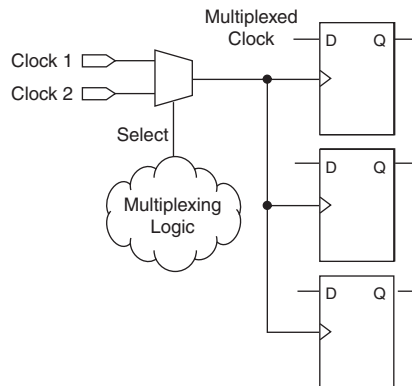
Ripple Counters

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. To make verification easier, avoid these types of structures.

Multiplexed Clocks

Clock multiplexing can be used to operate the same logic function with different clock sources. Multiplexing logic of some kind selects a clock source, as in Figure 6–5. For example, telecommunications applications that deal with multiple frequency standards often use multiplexed clocks.

Figure 6–5. Multiplexing Logic & Clock Sources



Adding multiplexing logic to the clock signal can lead to some of the problems discussed in the previous sections, but requirements for multiplexed clocks vary widely depending on the application. Clock multiplexing is acceptable if the following criteria are met:

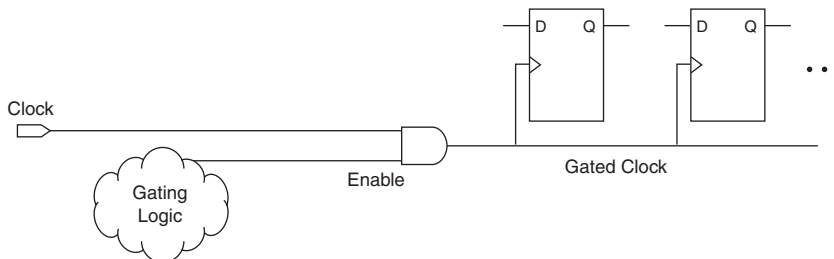
- The clock multiplexing logic does not change after initial configuration
- The design uses multiplexing logic to select a clock for testing purposes
- Registers are always reset when the clock switches
- A temporarily incorrect response following clock switching has no negative consequences

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.

Gated Clocks

Gated clocks turn a clock signal on and off using an enable signal that controls some sort of gating circuitry, as in [Figure 6–6](#). When a clock is turned off, the corresponding clock domain is shut down and becomes functionally inactive.

Figure 6–6. Gated Clock



Gated clocks can be a powerful technique to reduce power consumption. 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.

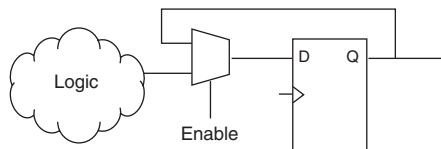
From a functional point of view, you can shut down a clock domain in a purely synchronous manner using a synchronous clock enable signal. However, when using a synchronous clock enable scheme, the clock network continues toggling. This practice does not reduce power

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. However, for improved power reduction, see “[Recommended Clock-Gating Method](#)” on page 6–11.

Synchronous Clock Enables

To turn off a clock domain in a synchronous manner, use a synchronous clock enable signal. Clock enable signals are efficiently supported in most FPGAs 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 6–7](#).

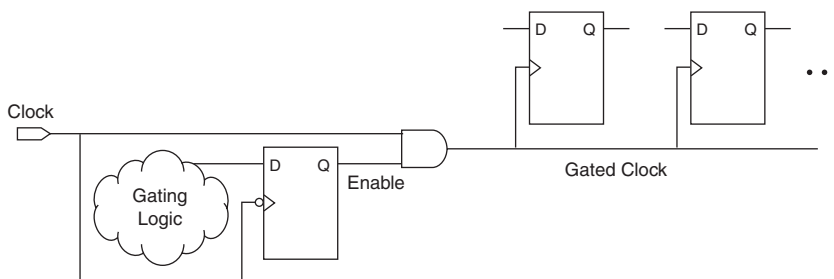
Figure 6–7. Synchronous Clock Enable



Recommended Clock-Gating Method

Use gated clocks only when your target application requires substantial power reduction. If you must use gated clocks, implement them using the robust clock-gating technique shown in [Figure 6–8](#).

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 6–8. Recommended Clock Gating Technique

In the technique shown in [Figure 6–8](#), a register generates the enable command to ensure that it 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 6–8](#)). Using this technique, only one input of the gate that turns the clock on and off changes at a time, which does not generate 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, being careful with the duty cycle and logic delay may be acceptable compared with the problems created by other methods of gating clocks.

Hierarchical Design Partitioning

A hierarchical design consists of multiple design blocks linked together in a hierarchy. When a design is partitioned hierarchically, you can 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.



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



For more information on hierarchical design flows, refer to the *Hierarchical Block-Based & Team-Based Design Flows* chapter in Volume 2 of the *Quartus II Handbook*.

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

- Partition the design at functional boundaries
- Minimize the I/O connections between different partitions
- Do not use “glue logic” or connection logic between hierarchical blocks. If 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.
- 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.
- Register all inputs and outputs of each block, which makes logic synchronous and avoids glitches. Also, registering outputs may eliminate the need to specify timing requirements for signals that connect between different blocks.

Targeting Clock & Register-Control Architectural Features

In addition to following general design guidelines, it is important to code your design with the target technology in mind. FPGAs provide device-wide clocks and register control signals that can improve performance.

Clock Network Resources

In ASIC design, balancing the clock delay as it is distributed across the device can be important. Altera FPGAs provide device-wide global clock routing resources and dedicated inputs, so there is no need to manually balance delays on the clock network. 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.

For best performance, limit the number of global clocks in your design to the number of dedicated global clock resources available in your FPGA. 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. PLL outputs can also drive the global and regional clock networks, and internal signals in the design can be routed onto the clock networks using the Global Signal logic option assignments in the Quartus II software.

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 certain devices, 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. 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. 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, refer to the *Recommended HDL Coding Styles* chapter in Volume 1 of the *Quartus II Handbook*.

Conclusion

Following the design practices outlined in this chapter can help you consistently meet your design goals. Asynchronous design techniques may result in incomplete timing analysis, may cause 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.

Introduction

Your hardware description language (HDL) coding style can have a big effect on the quality of results that you achieve for your Altera® design. 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. Designers are often in the best position to improve their 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 and 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/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. 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 may result in a less efficient result because the function will be 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 not easy 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 may want to 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.

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 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 chose 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/vvhd*. See [Table 7–1](#) for a list of generated files.



Altera strongly recommends that you use the wizard for complex megafunctions such as PLLs, transceivers, and LVDS drivers.

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 electronic design automation (EDA) synthesis tools. When implementing a megafunction with the clear box model, you provide the EDA 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.

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

[Table 7–1](#) lists and describes the MegaWizard Plug-In Manager-generated files.

Table 7–1. MegaWizard Plug-In Manager Generated Files (Part 1 of 2)

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

Table 7–1. MegaWizard Plug-In Manager Generated Files (Part 2 of 2)

File	Description
<output file>_inst.v (3)	Sample Verilog HDL instantiation of the module in the megafunction wrapper file.

Notes to Table 2–1:

- (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 will be created by default for most megafunctions. If you turn on the **Generate a clear box body (for EDA tools only)** option, the wizard will create a clear box netlist file to be 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.

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 will provide 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 memory, DSP blocks, and shift registers—that generally provide improved performance compared with regular logic.

This section describes the types of logic that standard synthesis tools recognize and map to megafunctions. Synthesis software infers only the specific functions listed in this section that are described by HDL code. 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.



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

Counters

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, and then convert the registers and logic to an `lpm_counter` megafunction. 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 (only to all logic value 1s)
- Asynchronous load
- Count enable
- Synchronous clear
- Synchronous set (only to all logic value 1s)
- Synchronous load
- Clock enable
- Up/down

The following code samples show simple Verilog HDL and VHDL counter function examples with different control signals.

Verilog HDL Counter with Count Enable & Asynchronous Clear

```
module counter (clk, reset, results, 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 Counter with Synchronous Load

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arth.ALL;
```

```
ENTITY count IS
  PORT (
    clock: IN STD_LOGIC;
    sload: IN STD_LOGIC;
    data:   IN STD_LOGIC_VECTOR (4 DOWNT0 0);
    result: OUT STD_LOGIC_VECTOR (4 DOWNT0 0)
  );
END count;

ARCHITECTURE rtl OF count IS
  SIGNAL result_reg : STD_LOGIC_VECTOR (4 DOWNT0 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;
      END IF;
    END IF;
  END PROCESS;

  result <= result_reg;
END rtl;
```

Adder/Subtractors

To infer adder/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. The software may then merge the adders and subtractors and convert them to an `lpm_addsub` megafunction.

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

Verilog HDL Adder/Subtractor

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

  reg [8:0] result;

  always @ (a or b or addnsub)
  begin
```

```
        if (addnsub)
            result = a + b;
        else
            result = a - b;
        end
    endmodule
```

VHDL 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;
```

Multipliers

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 website.

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 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 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 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 Signed Multiplier

```

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

Multiply-Accumulators & Multiply-Adders

Synthesis tools detect multiply-accumulators or multiply-adders and convert them to `altmult_accum` or `altmult_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 Unsigned Multiply-Accumulator with Input, Output &
Pipeline Registers (Latency = 3)*

```

module unsig_altmult_accum (dataout, dataa, datab, clk, aclr, clken);
    input [7:0] dataa;
    input [7:0] datab;
    input clk;
    input aclr;
    input clken;

    output [31:0] dataout;

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

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

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

    always @ (posedge clk or posedge aclr)
    begin
        if (aclr)
        begin
            dataa_reg <= 0;
            datab_reg <= 0;

            multa_reg <= 0;

            dataout <= 0;
        end

        else if (clken)
        begin
            dataa_reg <= dataa;
            datab_reg <= datab;

            multa_reg <= multa;

            dataout <= adder_out;
        end
    end
endmodule

```

Verilog HDL Signed Multiply-Adder (Latency = 0)

```
module sig_altmult_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_signed.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 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;
USE ieee.std_logic_unsigned.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;
END rtl;

```

```
accum_out <= std_logic_vector(adder_out);  
END rtl;
```

RAM

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 the 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, the Quartus II Help explains the condition under which the functionality changes.

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 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 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 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 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 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 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;

        q <= ram_block[read_address];
    end
endmodule
```

VHDL 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 DOWNT0 0);
    write_address IN STD_LOGIC_VECTOR (ADDRESS_WIDTH - 1 DOWNT0 0);
    read_address IN STD_LOGIC_VECTOR(ADDRESS_WIDTH - 1 DOWNT0 0);
    we : IN STD_LOGIC;
    q : OUT STD_LOGIC_VECTOR(DATA_WIDTH - 1 DOWNT0 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 DOWNT0 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;

```

ROM

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.

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, 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 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 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
        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 PROCESS;
END rtl;

```

Shift Registers

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

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 for different sizes of shift registers, external decode logic (using LEs or ALMs) is 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 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
endmodule
```

```

        end
    end

    assign sr_out = sr[63];
endmodule

```

The following code sample shows a Verilog HDL 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 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

```

The following code sample shows a VHDL example of a 8-bit wide, 64-bit long shift register with evenly spaced taps at 15, 31, and 47.

VHDL 8-Bit Wide, 64-Bit Long Shift Register with Evenly Spaced Taps

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY shift_8x64_taps IS
  PORT (
    clk :      IN STD_LOGIC;
    shift :    IN STD_LOGIC;
    sr_in :    IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    sr_tap_one : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    sr_tap_two : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    sr_tap_three : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    sr_out :   OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
END shift_8x64_taps;

ARCHITECTURE arch OF shift_8x64_taps IS

  SUBTYPE sr_width IS STD_LOGIC_VECTOR(7 DOWNTO 0);
  TYPE sr_length IS ARRAY (63 DOWNTO 0) OF sr_width;

  SIGNAL sr : sr_length;
BEGIN
  PROCESS (clk)
  BEGIN
    IF (clk'EVENT and clk = '1') THEN
      IF (shift = '1') THEN
        sr(63 DOWNTO 1) <= sr(62 DOWNTO 0);
        sr(0) <= sr_in;
      END IF;
    END IF;
  END PROCESS;

  sr_tap_one <= sr(15);
  sr_tap_two <= sr(31);
  sr_tap_three <= sr(47);
  sr_out <= sr(63);
END arch;
```

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 Flip-Flops

FPGA device architectures are based on registers, or flip-flops. The registers in Altera FPGAs provide a number of secondary control 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.



Note that 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 has been 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.

Note that **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 HDL D-Flip-Flop (Register) with Control Signals

```
module dff_control( clk, aclr, aload, ena, data, adata,
q );
    input clk, aclr, aload, ena, data, adata;
    output q;
    reg q;
    always @ (posedge clk or posedge aclr or posedge
aload)
        begin
            if (aclr)
                q <= 1'b0;
            else if (aload)
                q <= adata;
            else
                if (ena)
                    q <= data;
        end
endmodule
```

VHDL D-Flip-Flop (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, ena, 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.



Note that in hierarchical or block-based designs, a hierarchical boundary can not contain any bidirectional ports.

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

Tri-State Signal in Verilog HDL

```
module tristate (myinput, myenable, mybidir);
    input myinput, myenable;
    inout mybidir;

    assign mybidir = (myenable ? myinput : 1'bZ);
endmodule
```

Tri-State Signal in VHDL

```
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 lookup 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 (LEs)

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 Pipelined Binary Tree

```
module binary_adder_tree (A, B, C, D, E, CLK, OUT);
    parameter WIDTH = 16;

    input [WIDTH-1:0] A, B, C, D, E;
    input CLK;
    output [WIDTH-1:0] OUT;

    wire [WIDTH-1:0] sum1, sum2, sum3, sum4;

    reg [WIDTH-1:0] sumreg1, sumreg2, sumreg3, sumreg4;

    // Registers
    always @ (posedge CLK)
        begin
            sumreg1 <= sum1;
            sumreg2 <= sum2;
            sumreg3 <= sum3;
            sumreg4 <= sum4;
        end

    // 2-bit additions
    assign sum1 = A + B;
    assign sum2 = C + D;
    assign sum3 = sumreg1 + sumreg2;
    assign sum4 = sumreg3 + E;

    assign OUT = sumreg4;
endmodule
```

Architectures With Six-Input LUTs in Adaptive Logic Modules (ALMs)

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 more streamlined coding style. Specifically, Stratix II device ALMs can simultaneously add three bits. Thus, the tree in the previous example need be only 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. While Quartus II integrated synthesis reports that 32 ALUTs are used to implement the function, the Quartus II Fitter may report a slightly higher number. However, in a typical design, the Quartus II Fitter can pack other logic into the LAB to take advantage of the unused ALUTs.

Verilog HDL Pipelined Ternary Tree

```
module ternary_adder_tree (A, B, C, D, E, CLK, OUT);
    parameter WIDTH = 16;

    input [WIDTH-1:0] A, B, C, D, E;
    input CLK;
    output [WIDTH-1:0] OUT;

    wire [WIDTH-1:0] sum1, sum2;

    reg [WIDTH-1:0] sumreg1, sumreg2;

    // Registers
    always @ (posedge CLK)
        begin
            sumreg1 <= sum1;
            sumreg2 <= sum2;
        end

    // 3-bit additions
```

```

assign sum1 = A + B + C;
assign sum2 = sumreg1 + D + E;

assign OUT = sumreg2;
endmodule

```

These examples apply to 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 `sum = (A + B + C) + (D + E)` is more likely to create the optimal implementation of a 3-input adder for `A + B + C` followed by a 3-input adder for `sum1 + D + E` than the code without the 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.

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 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 7–33](#) and [“VHDL State Machines” on page 7–36](#).

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 the state `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.

Example State Machine in Verilog HDL

```

module verilog_fsm (clk, reset, in_1, in_2, out);
    input clk;
    input reset;
    input [3:0] in_1;
    input [3:0] in_2;

    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_0` instead of a `state_0`, 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 `in_1` and `in_2` is used as an output of the state machine in the states `state_1` and `state_2`. The difference (`in1 - in2`) is also used in the states `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 `in_1` and `in_2`. The use of these temporary variables in the various states of the state machine ensures the proper resource sharing between these mutually exclusive states.

Example State Machine in VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
```



```

ENTITY vhdl_fsm IS
  PORT(
    clk: IN STD_LOGIC;
    reset: IN STD_LOGIC;
    in1: IN STD_LOGIC_VECTOR(4 downto 0);
    in2: IN STD_LOGIC_VECTOR(4 downto 0);
    out_1: OUT STD_LOGIC_VECTOR(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 tmp_out_0: STD_LOGIC_VECTOR (4 downto 0);
  SIGNAL tmp_out_1: STD_LOGIC_VECTOR (4 downto 0);
  SIGNAL state: Tstate;
  SIGNAL next_state: Tstate;

BEGIN
  PROCESS(clk, reset)
  BEGIN
    IF reset = '1' THEN
      state <= state_0;
    ELSIF rising_edge(clk) THEN
      state <= next_state;
    END IF;
  END PROCESS;

  PROCESS (state, in1, in2, tmp_out_0, tmp_out_1)
  BEGIN
    tmp_out_0 <= STD_LOGIC_VECTOR'(UNSIGNED(in1)+UNSIGNED(in2));
    tmp_out_1 <= STD_LOGIC_VECTOR'(UNSIGNED(in1)+UNSIGNED(in2));

    CASE state IS
      WHEN state_0 =>
        out_1 <= in1;
        next_state <= state_1;
      WHEN state_1 =>
        IF (in1 < in2) then
          next_state <= state_2;
          out_1 <= tmp_out_0;
        ELSE
          next_state <= state_3;
          out_1 <= tmp_out_1;
        END IF;
      WHEN state_2 =>
        IF (in1 < "0100") then
          out_1 <= tmp_out_0;
        ELSE
          out_1 <= tmp_out_1;
        END IF;
      next_states <= state_3;
    END CASE;
  END PROCESS;

```

```
    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 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 look-up tables (LUTs) 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 “[Simple Binary-Encoded “Case” Statement](#)” example below shows Verilog HDL code that describes a simple 4:1 binary multiplexer.

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. “Simple One-Hot-Encoded “Case” Statement” example below shows a simple Verilog HDL code samples that describes a one-hot selector multiplexer.

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 “IF Statement Implying Priority” example below is likely to result in the implementation illustrated schematically in Figure 7–1.

IF Statement Implying Priority

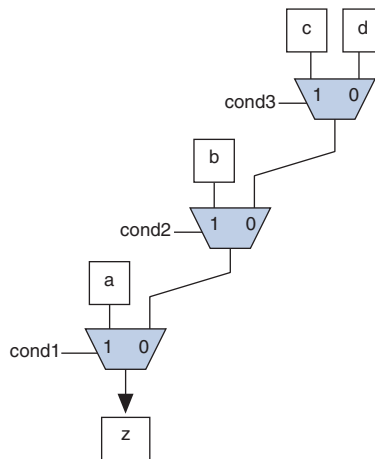
```

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 7-1 form a chain, evaluating each condition, or select bit, one at a time.

Figure 7-1. Priority Multiplexer Implementation of the IF Statement in “IF Statement Implying Priority” on page 7-40



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 “[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).

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;
```

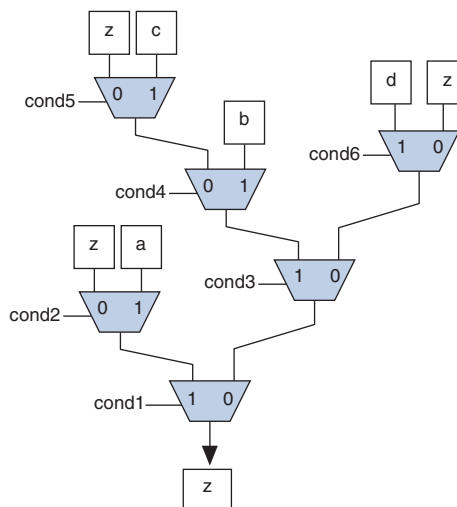
However, each of the three separate IF statements in the code has an implicit ELSE condition that is not specified. Since the output values for the ELSE cases are not specified, the synthesis tool assumes the intent is to maintain the same output value for these cases. The code sample in the [“IF Statement with Default Conditions Explicitly Specified”](#) example shows code with the same functionality as the code in the [“IF Statement with Implicit Defaults”](#) on page 7–42 example but specifies the ELSE cases explicitly.

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 7–2](#) is a schematic representation of the code in the [“IF Statement with Default Conditions Explicitly Specified”](#) example above, illustrating that although there are only four inputs, the multiplexing logic is significantly more complicated than a basic 4:1 mux.

Figure 7-2. Multiplexer Implementation of the IF Statements in “IF Statement with Implicit Defaults” on page 7-42 and “IF Statement with Default Conditions Explicitly Specified” on page 7-42



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

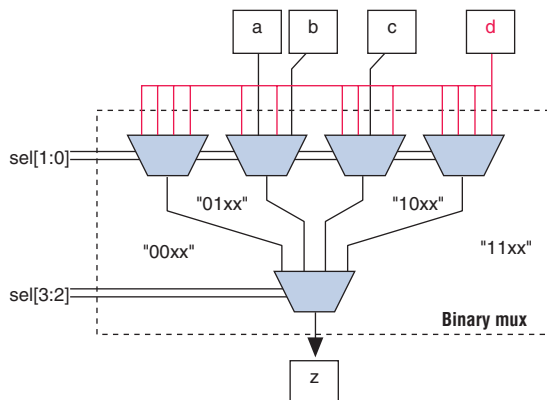
A degenerate multiplexer is one in which not all of the possible cases are used for unique data inputs. The unneeded cases tend to contribute to inefficiency in the logic utilization for these multiplexers. You can recode degenerate muxes so that they take advantage of the efficient logic utilization 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 7-3 on page 7-44](#) 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.

CASE Statement Describing a Degenerate Multiplexer

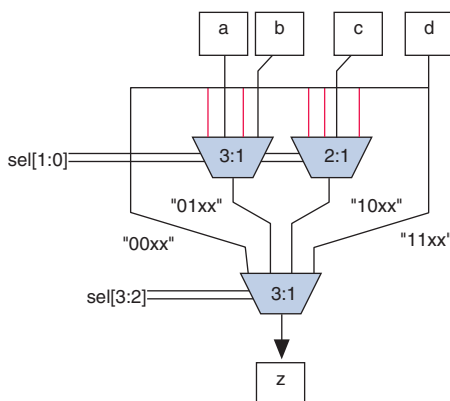
```
CASE sel[3:0] IS
  WHEN "0101" => z <= a;
  WHEN "0111" => z <= b;
  WHEN "1010" => z <= c;
  WHEN OTHERS => z <= d;
END CASE;
```

Figure 7-3. Binary Multiplexer Implementation of “CASE Statement Describing a Degenerate Multiplexer” on page 7-44



In the example in [Figure 7-3](#), 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 7-4](#).

Figure 7-4. Optimized Version of the Degenerate Binary Multiplexer from Figure 7-3



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. “[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 “[4:1 Binary Multiplexer Design](#)” below provides code to implement the full binary mux.

Recoder Design for Degenerate Binary Multiplexer

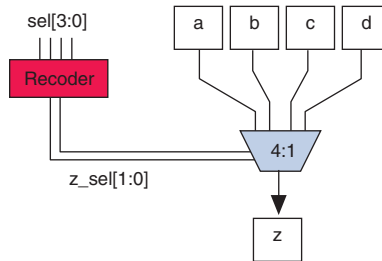
```
CASE sel[3:0] IS
    WHEN "0101" => z_sel <= "00";
    WHEN "0111" => z_sel <= "01";
    WHEN "1010" => z_sel <= "10";
    WHEN OTHERS => z_sel <= "11";
END CASE;
```

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 7-5](#). 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 7-5. Binary Multiplexer with Recorder



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

The 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 Quartus II 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 software.

This section includes the following chapters:

- Chapter 8, Quartus II Integrated Synthesis
- Chapter 9, Synplicity Synplify & SynplifyPro Support
- Chapter 10, Mentor Graphics LeonardoSpectrum Support
- Chapter 11, Mentor Graphics Precision RTL Synthesis Support
- Chapter 12, Synopsys FPGA Compiler II BLIS & Quartus II LogicLock Design Flow
- Chapter 13, Synopsys Design Compiler FPGA Support
- Chapter 14, Analyzing Designs with the Quartus II RTL Viewer & Technology Map Viewer

Revision History

The table below shows the revision history for [Chapters 8 to 14](#).

Chapter(s)	Date / Version	Changes Made
8	June 2004 v2.0	<ul style="list-style-type: none"> • Updates to tables, figures. • New functionality for Quartus 4.1.
	Feb. 2004 v1.0	Initial release.
9	June 2004 v2.0	<ul style="list-style-type: none"> • Updates to tables, figures. • New functionality for Quartus 4.1.
	Feb. 2004 v1.0	Initial release.
10	June 2004 v2.0	<ul style="list-style-type: none"> • Updates to tables, figures. • New functionality for Quartus 4.1.
	Feb. 2004 v1.0	Initial release.
11	June 2004 v2.0	<ul style="list-style-type: none"> • Updates to tables, figures. • New functionality for Quartus 4.1.
	Feb. 2004 v1.0	Initial release.
12	June 2004 v1.0	No change to document.
	Feb. 2004 v1.0	Initial release.
13	June 2004 v1.0	Initial release.
14	June 2004 v 2.0	<ul style="list-style-type: none"> • Updates to tables, figures. • New functionality for Quartus 4.1.
	Feb. 2004 v1.0	Initial release.

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 HDL support in the Quartus II software, and explains how to improve and control your Quartus II synthesis results by using Quartus II synthesis options, setting other Quartus II options in your Verilog HDL or VHDL source code, and controlling the interface of architecture-specific megafunctions.

Verilog HDL & VHDL Support



This section explains the Quartus II software's integrated synthesis support for the Verilog HDL and VHDL synthesizable language features, as well as some synthesis directives and attributes.

For information on specific syntax features and language constructs, see *Quartus II Verilog HDL Support* and *Quartus II VHDL Support* in Quartus II Help. Quartus II Help also describes the full support for Altera Hardware Description Language (AHDL) Text Design Files (**.tdf**) and schematic entry Block Design Files (**.bdf**), as well as how to import Graphical Design Format (**.gdf**) files from the MAX+PLUS® II software. These Altera-specific file formats are not described in this chapter.

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.

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

Unsupported Verilog-2001 standard constructs include the following:

- Libraries and configurations



See Quartus II Help for a complete listing of supported constructs.

The Quartus II software supports case-sensitive Verilog HDL code, in accordance with the Verilog HDL standard.

The Quartus II software supports the ``include` construct 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.

The Quartus II software supports VHDL libraries differently from the MAX+PLUS® II software or versions of the Quartus II software earlier than version 2.1. In the Quartus II software version 2.1 and later, standard IEEE and vendor VHDL libraries and packages can be called from VHDL code in the Quartus II software.

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`, `altera_mf_components`, and `lpm_components` in the ALTERA library



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

The Quartus II software does not support user-defined precompiled libraries.

To call a user-defined VHDL package in the Quartus II software, specify the library and package name using the `LIBRARY` and `USE` commands. You can use any name for your library, including `work`; therefore, you can use current software versions for projects developed with older versions of Altera software that used precompiled libraries without the need to modify any code. To compile using a VHDL package projects, include the VHDL package in your Quartus II project on the **Files** page of the **Settings** dialog box (Assignments menu). The package must be listed before other files that use the package because it must be analyzed by the Quartus II Compiler first.

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](#)”, 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.

Verilog HDL

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

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 in this chapter. The examples demonstrate each syntax form.

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.

The Verilog-2001 and VHDL language definitions provide specific syntax for specifying attributes. However in Verilog-1995 HDL, you must use 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.

Verilog-1995 HDL

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



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 8–11 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.

Verilog-2001 HDL

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

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 the “[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;
```

VHDL

attribute <attribute> : <attribute type> ;

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

In this chapter, the examples demonstrate each syntax form.

Assignments or settings made with synthesis attributes take precedence over assignments or settings made through the Quartus II user interface, the Quartus Settings File (**.qsf**), and the Tcl interface.

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 (GUI) 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, 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 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 8–23.



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 the 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.

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. 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*. VHDL case statements must be full, so the attribute does not apply.

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`.

Sample Verilog HDL Code with a `full_case` Attribute

```
module full_case (a, sel, y);
    input [3:0] a;
    input [1:0] sel;
    output y;
    reg y;
    always @ (a or sel)
        case (sel) // synthesis full_case
            2'b00: y=a[0];
            2'b01: y=a[1];
            2'b10: y=a[2];
        endcase
endmodule
```

Verilog-2001 syntax also accepts the 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 statements in Verilog HDL case statements may overlap. To resolve multiple matching case items, the Verilog 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 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 generation logic (allowing implementations such as multiplexing logic instead of a priority encoder).

In VHDL, the individual case items 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`) will be high as well, a situation that cannot occur in functional HDL simulation.

Sample Verilog HDL Code with 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

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

Verilog-2001

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

VHDL

```
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 removes 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

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

Verilog-2001

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

VHDL

```
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 PowerFit™ 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 re-timing.



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

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

Verilog-2001

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

VHDL

```
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 8–1](#) 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 8–1. 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	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.

The default setting varies by target device family, and is generally optimized for the best area/speed trade-off. Results are design-dependent and can 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 8–2](#) 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 8–2. 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.
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 complex programmable logic devices (CPLDs). These settings achieve the best results on average, but another encoding style might be more appropriate for your design, so these options allow 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 8–3](#) 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 8–3. 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.

Restructure Multiplexers

This option specifies whether the Quartus II software should extract and optimize buses of muxes 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 Cyclone™, Cyclone II, MAX II, Stratix®, Stratix GX, and Stratix 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 clock speed, f_{MAX} .

Table 8–4 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 8–4. 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 Off 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 8–5 describes the information that is listed in the **Multiplexer Restructuring Statistics** report table for each bus of multiplexers.

Table 8–5. 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.

Table 8–5. Multiplexer Information in the Multiplexer Restructuring Statistics Report

Heading	Description
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 makes 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 like 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 8–6](#) 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 only recognizes this option 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 8–6](#). You can apply this option to an individual register or a design entity that contains registers.

Table 8–6. 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 using the **Assignment Editor** (Assignments menu).



Any registers that the software maps to the `altmult_accum` and `altmult_add` megafunctions and places in DSP blocks are not available in the Simulator because their node names do not exist after synthesis.

Shift Registers

Use the **Auto Shift Register Replacement** logic option to control shift register inference. This option is turned on by default. To disable inference, turn off this option for your whole project on the **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu), or for a specific block using 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.

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 using 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.

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 embedded memory block.

Sample Verilog-1995 Code with a ramstyle Attribute

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

Sample Verilog-1995 Code with a ramstyle Attribute

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

Sample VHDL Code with 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";
```

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](#)” 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

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.

Sample Verilog HDL Code with 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;
```

Sample VHDL Code with 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 & Synthesis, such as Fitting. The syntax for setting this attribute is the syntax defined in the section [“Synthesis Attributes” on page 8–5](#) for HDL attributes (examples are provided below).

Assignments and settings made with the Altera Attribute take precedence over assignments and settings made through the Quartus II user interface, the Quartus Settings File (`.qsf`), and the Tcl interface.

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

```
"<variable_1>=<value_1>;<variable_2>=<value_2>[ ; ... ]"
```

If the Quartus II option or assignment includes a target, source, and/or section tag, you can use the following syntax (similar to the syntax of the QSF file) before the QSF assignment variable and value:

```
{ -from "<source>" -to "<target>" -section_id "<section>" }
```

Each of the tags above is optional, but if any tags are included then you need to use the braces {}. The syntax for the full attribute value, including the optional target, source, and section tags for two different QSF assignments is as follows:

```
"[{ [-from "<source_1>"] [-to "<target_1>"] [-section_id "<section_1>"] ]}  
<variable_1>=<value_1>; [{ [-from "<source_2>"] [-to "<target_2>"] [  
section_id "<section_2>"] ]} <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 file.

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 = "POWER_UP_LEVEL=HIGH" */;
```

Verilog-2001 Example of Applying Altera Attribute to an Instance

```
(* altera_attribute = "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 "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 =
"AUTO_SHIFT_REGISTER_RECOGNITION=OFF" */;
```

Verilog-2001 Example of Applying Altera Attribute to an Entity

```
(* altera_attribute = "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 "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" */;
```

Verilog-2001 Example of Applying Chip Pin to Part of a Bus of Pins

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

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";
```

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> -to <Instance Name>
```

Quartus II Synthesis Options

Table 8–7 lists the QSF variable name and applicable values for the settings discussed in this chapter. The QSF variable name is used in the Tcl assignment to make the setting along with the appropriate value. The Type column indicated whether the setting is supported as a Global setting, or an Instance setting, or both.

Table 8–7. Quartus II Synthesis Options			
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

Table 8–7. Quartus II Synthesis Options

Setting Name	QSF Variable	Values	Type
State Machine Processing	STATE_MACHINE_PROCESSING	AUTO “MINIMAL BITS”, “ONE HOT”, “USER-ENCODED”	Global, Instance
Optimization Technique	<device name>_OPTIMIZATION_TECHNIQUE	Area, Speed, Balanced	Global, Instance
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.

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. This document describes methodologies that you can use to improve synthesis results and obtain optimum performance in your target Altera device.

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 HDL code
- Block-based design with the Quartus II LogicLock™ methodology, including the SynplifyPro 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

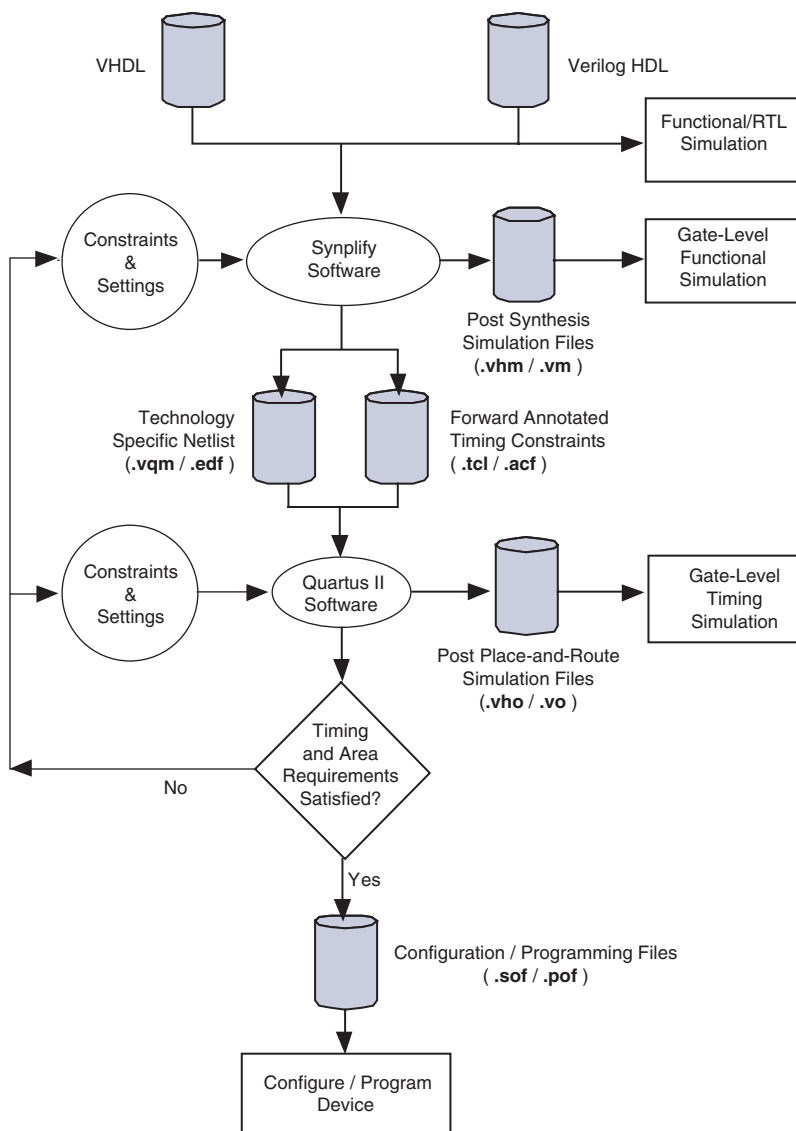
The basic steps in a Quartus II design flow using the Synplify software are the following:

1. Create Verilog HDL and/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 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 9–1 shows the recommended design flow when using the Synplify and Quartus II software.

Figure 9–1. 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) by using the SCOPE editor in the Synplify software or the HDL source file. Compiler directives can also be defined in the HDL source file. Many of these constraints are forward-annotated for use by the Quartus II software in the Tcl file. You can save all project options and included files in a Synplify project file (.prj).

The HDL Analyst included in the Synplify software is a graphical tool for generating schematic views of the technology-independent RTL view netlist (.srs) and technology-view netlist (.srm) files. You can 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 9–9](#).



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 9–1](#), 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 produce several intermediate and output files. [Table 9–1](#) lists these files with a short description of each file.

Table 9–1. Synplify Intermediate & Output Files

File Extensions	File Description
.srs	Technology independent register transfer level (RTL) netlist that can be read only by Synplify
.srm	Technology view netlist
.vm/.vhm	Post-synthesis output design file in Verilog HDL/VHDL format that you can use for post-synthesis simulation
.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 9–1](#)

- (1) This report file includes performance estimates which 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 only created 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 only created for ACEX 1K, FLEXR 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 tool command language (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 Altera 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 labelled 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 of chapter 3: *Tasks and Tips* in the *Synplify User Guide*.

Implementations in Synplify Pro

Use the **New Implementation** option (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 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 to 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 by default. 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 TCO and TSU values directly to inputs and outputs. However, a TCO value can be inferred by setting an external output delay, and a TSU value can be inferred by setting an external input delay. The following equations illustrate the relationship between TCO/TSU and the input/output delays:

$$\text{TCO} = \text{Clock period} - \text{external output delay}$$
$$\text{TSU} = \text{Clock period} - \text{external input delay}$$

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 through 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 and/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 9-2](#).

Table 9-2. <i>syn_encoding</i> 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 as much of a concern.
Gray	Generates state machines where only one flip-flop changes during each transition. Gray-encoded state machines tend to be glitchless.
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 9–11 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 automatically 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_retiming` attribute.

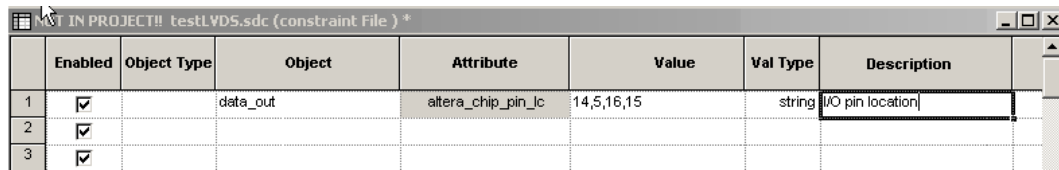
Altera Specific Attributes

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 9–2](#) shows how to set the attribute in the SCOPE editor.

Figure 9–2. *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"/>						

[“altera_chip_pin_lc with VHDL for ACEX 1K and FLEX 10KE Devices”](#) 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";
```

[“altera_chip_pin_lc with Other Devices”](#) 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 this attribute to implement logic in ESB/EABs rather than logic resources to improve area utilization. The modules 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 ESB/EABs only. For example, the Stratix® architecture is not supported by this option. For designs targeting devices with no ESB/EABs, this has no effect.)

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 **.edf**) 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 places the synthesized output file, forward-annotated timing constraints, and pin assignments in a named Quartus II project. 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 through Natvelink 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 executing 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 then 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 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.



For more information on specific Altera megafunctions, see the Quartus II Help. For more information on IP functions, consult the appropriate IP documentation.

If you decide 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 9–18](#) describes the MegaWizard flow with the Synplify software

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 9–23](#).



For a detailed discussion on instantiating vs. inferring on 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 to set up and parameterize a megafunction, the MegaWizard 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-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 take better advantage of timing driven optimization than a black box methodology.

This clear box feature of the MegaWizard can be turned on by checking the **Generate clear box body (for EDA tools only)** in the **MegaWizard Plug-In Manager** (Tools menu) for certain megafunctions. If the option above 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 on this option causes the Quartus II MegaWizard to generate a synthesizable clear box netlist instead of the megafunction wrapper file described in the “**Black Box Methodology**” on page 9–19.

Using MegaWizard-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 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-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 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-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 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. See [“Other Synplify Software Attributes for Black-boxing” on page 9–22](#) for details.

Using MegaWizard-generated Files for Certain LPM Functions in the Synplify LPM timing flow

For the LPM_MULT, LPM_RAM_DP, LPM_RAM_DQ, LPM_ROM, LPM_LATCH, and LPM_FF megafunctions, you don't need take any steps to manually create a black-box declaration. The Synplify LPM timing flow allows you to directly instantiate these LPMS in your Synplify project. By directly instantiating these LPMS, the Synplify software can estimate timing for these functions during synthesis using Altera-provided timing models.



For other megafunctions, follow the black-box methodologies described in the following sections.

For Verilog HDL designs, include the **altera_lpm.v** library file from the lib/altera directory of the Synplify installation directory. Include the Megafunction variation wrapper file *<output file>.v* or *<output file>.vhd* generated by the MegaWizard in the Synplify project, and compile it as a normal block in your design.

Using MegaWizard-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 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 along with your Synplify-generated VQM netlist in 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 9–22.

“Top-level Verilog HDL Code with Black Box Instantiation of LPM_COUNTER” below shows a sample top-level file that instantiates **verilogCounter.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-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 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 in your Quartus II project.

You can use the `syn_black_box` compiler directive 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 DOWNT0 0)
    );
END testCounter;
ARCHITECTURE top OF testCounter IS
    component vhdlCount
    PORT (
        clock: IN STD_LOGIC ;
        q: OUT STD_LOGIC_VECTOR (7 DOWNT0 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 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 [“Verilog HDL Example” on page 9–22](#) 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_tpd1="addr[3:0]->z[3:0]=8.0"  
   syn_tsu1="addr[3:0]->clk=2.0"  
   syn_tsu2="we->clk=3.0" */  
output[3:0]z;  
input[3:0]d;  
input[3:0]addr;  
input we  
input clk  
endmodule
```

The following other attributes are supported by the Synplify 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 of Chapter 3: *Tasks and Tips in 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.

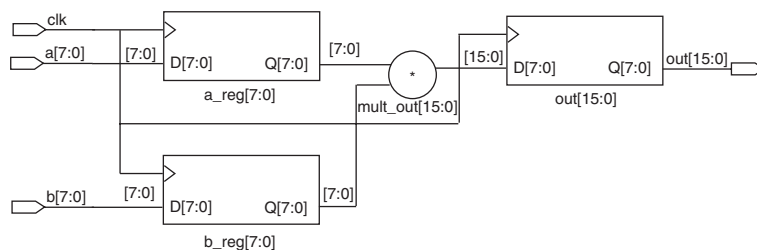


For coding style recommendations and examples for inferring megafunctions in Altera devices, see the *Recommended HDL Coding Styles* chapter in Volume 1 of the *Quartus II Handbook*.

Multipliers

Figure 9–3 provides 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 `lpm_mult` megafunction. For devices with DSP blocks, the software may implement the `lpm_mult` function in a DSP block instead of LEs, depending on device utilization.

Figure 9–3. 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 implies a fixed number of embedded multipliers. If the design uses more multipliers than are available, then the Synplify software automatically maps the extra multipliers to LEs.

If a design has 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/or multipliers that are not in the critical paths may then be implemented in LEs. 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 elements 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 9–3 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 9–3. Attribute Settings for DSP Block in the Synplify Software

Attribute Name	Value	Description
<code>syn_multstyle</code>	<code>lpm_mult</code>	LPM Function inferred and multipliers implemented in DSP block
<code>syn_multstyle</code>	<code>logic</code>	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. See your 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 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;
```

“VHDL Code for Inferred Dual-Port RAM Preventing Bypass Logic” on page 9–27 shows sample 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 SynplifyPro 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 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 *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*. For more information on hierarchical design methodologies and design flows using the Quartus II software, see *Hierarchical Block-Based and Team-Based Design Flow* chapter in Volume 1 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 project(s) to place-and-route the design.

Creating a Design with Multiple VQM Files using Multipoint Synthesis (Synplify Pro only)

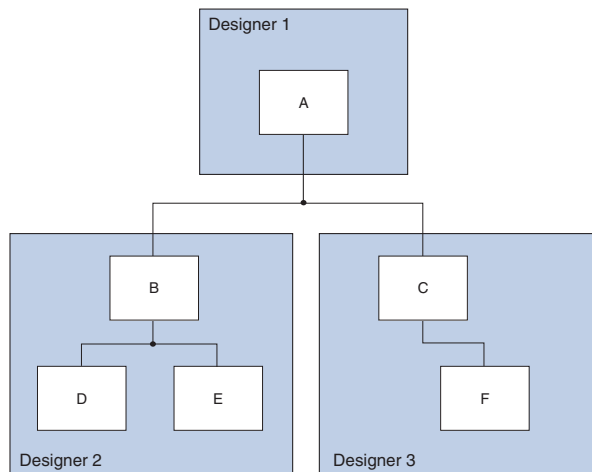
This section describes how to generate multiple VQM files using the Synplify Pro MultiPoint synthesis flow. You must first set up your compile points, constraint files, and Synplify Pro options, then apply Altera-specific attributes to create LogicLock regions.

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 9–4 shows an example of a design hierarchy that can be split into multiple compile points.

Figure 9–4. Design Hierarchy



In this case, modules A, B, and C are considered 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 is 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. 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-states coded in lower-level files do not get automatically pushed to the top-level.
- 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 9–4 shows the valid combinations of the LogicLock attributes.

Table 9–4. LogicLock Location and Size Properties		
altera_logiclock_location Attribute	altera_logiclock_size Attribute	Description
Floating	Auto	The most flexible kind 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 `cp1l_1` was selected by Synplify Pro for the compile point.

```
project add_assignment "top" "cp1l_1" "" "" "LL_AUTO_SIZE" "ON"
project add_assignment "top" "cp1l_1" "" "" "LL_STATE" "FLOATING"
project add_assignment "top" "cp1l_1" "" "|A:u1" "LL_MEMBER_OF" "cp1l_1"
```

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

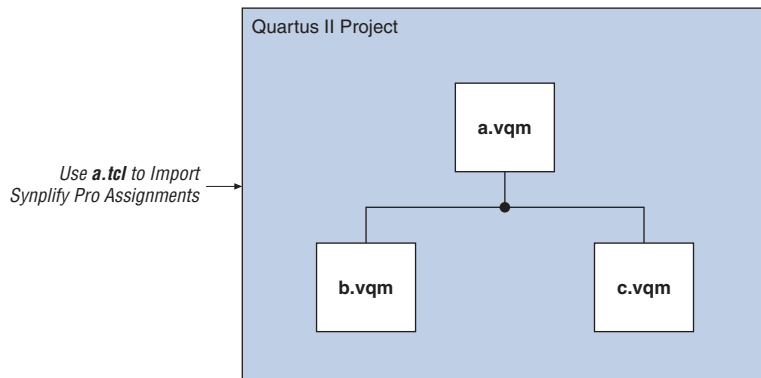
You can use the following methods to import the VQM files into the Quartus II software.

- Use the *<top-level project>.tcl* file that contains the SynplifyPro 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 9–5 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.

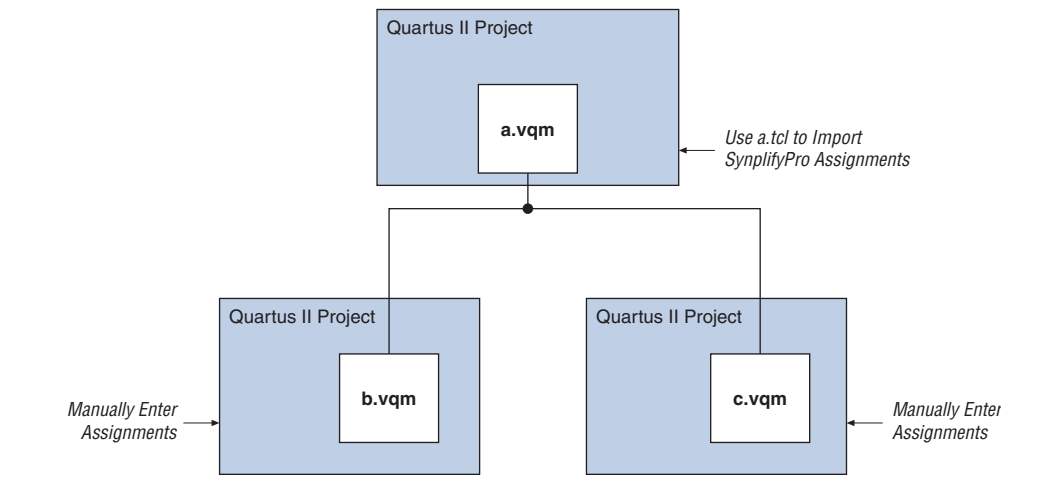
Figure 9–5. Design Flow Using Multiple VQM Files with One Quartus II Project



- 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 9–6 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 has to manually enter their assignments into the Quartus II software because Synplify Pro doesn't create a Tcl file for the lower-level modules.

Figure 9–6. 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 the Synplify software that does 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:

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:

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 sections below describe an example of black-boxing modules using the files described in *Hierarchical Block-Based & Team-Based Design Flows* chapter in Volume 1 of the *Quartus II Handbook*. 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/.vhd, D.v/.vhd, and E.v/.vhd as the source files.
2. Generate a VQM file for module C. Use C.v/.vhd and F.v/.vhd as the source files.
3. Generate a top-level VQM 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, 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.

“Black-Boxing Example for Top-Level File A.v” 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 [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

```

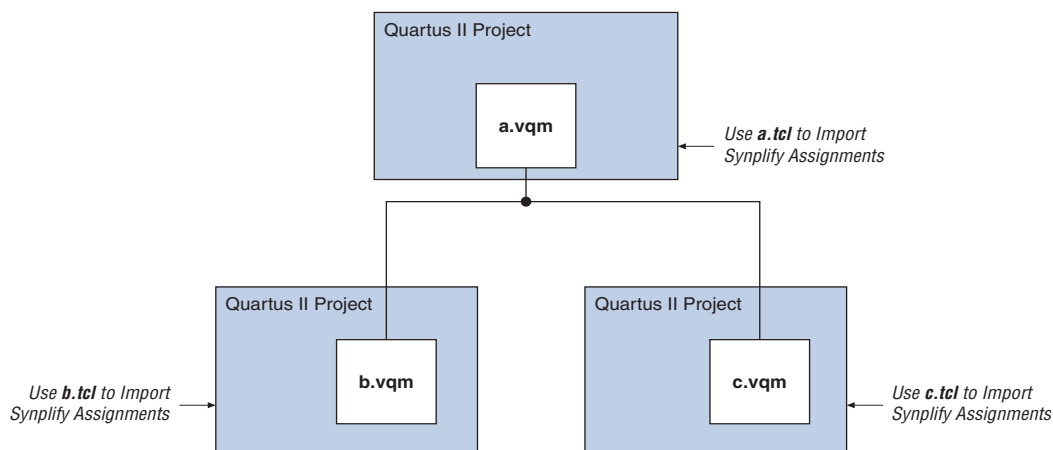
```
PORT MAP (  
    d => cnt_out,  
    clk => clk,  
    e => e,  
    q => data_out );  
  
-- Any other code in A.vhd goes here  
  
END a_arch;
```

After you have completed the steps outlined in this section, you will have different VQM 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 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 9-7](#) 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 9–7. 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.

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. Combining hardware description language (HDL) coding techniques, Mentor Graphics® LeonardoSpectrum™ software constraints, and Altera® Quartus® II software options can 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 software 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 <http://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:

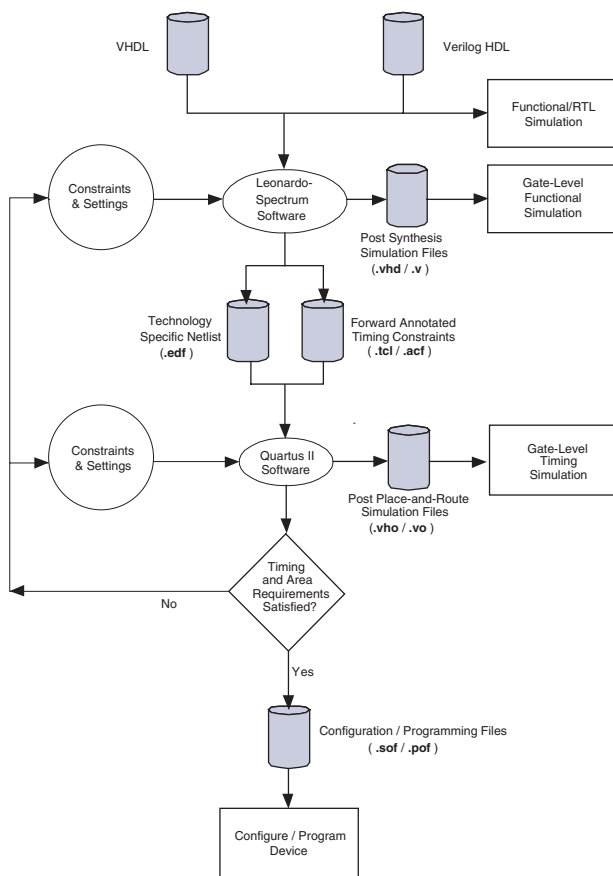
1. Create Verilog HDL or VHDL design files in the LeonardoSpectrum software or a text editor.
2. Import the Verilog HDL or VHDL design files to the LeonardoSpectrum software for synthesis.
3. Select a target device and add timing constraints and compiler directives to help optimize the design during 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.

- After obtaining place-and-route results that meet your needs, configure or program the Altera device.

Figure 10–1 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 10–1, 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 10–1. 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 10–1 lists these file extensions with a short description of each file.

Table 10–1. LeonardoSpectrum Intermediate & Output Files	
File Extension(s)	File Description
<code>.xdb</code>	Technology independent register transfer level (RTL) netlist file that can only be read by the LeonardoSpectrum software
<code>.v/.vh</code>	Post-synthesis output design file in Verilog HDL and VHDL format that you can use for post-synthesis simulation
<code>.edf</code>	Technology-specific output netlist in electronic design interchange format (EDIF)
<code>.acf/.tcl (1)</code>	Forward-annotated constraint file containing constraints and assignments

Note to Table 10–1:

- (1) An assignment and configuration file (`.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 tool command language (Tcl) file (`.tcl`) is generated for the Quartus II software which also contains Tcl commands to create a Quartus II project.



Altera recommends that you do not use project directory names that includes spaces. Some file operations in the LeonardoSpectrum software may 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 result. 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 different types of timing constraints in LeonardoSpectrum.

The timing constraints described in the “**Global Power Tab**” section can be set in the **Constraints** Flow tab. In this tab, there are different Power tabs at the bottom, such as **Global** and **Clock**, for setting the different 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

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 to individual input or output pins in the **Input** and **Output** tabs. First, select a name from 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 10–2 shows the state machine encoding styles supported by the LeonardoSpectrum software.

Table 10–2. State Machine Encoding Styles in the LeonardoSpectrum Software	
Style	Description
Binary	Generates state machines with the fewest possible flip-flops. Binary state machines are useful for area-critical designs when timing is not as much of a concern.
Gray	Generates state machines where only one flip-flop changes during each transition. Gray-encoded state machines tend to be glitchless.
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.
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 made 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 FPGA devices containing I/O cells or I/O elements. 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

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 10–2 shows an example of a LeonardoSpectrum compilation report.

Figure 10–2. 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
```

LeonardoSpectrum software estimates the timing results based on timing models. The LeonardoSpectrum software does not know 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 from 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 10–9 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®-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 Via 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. Open the Quartus II Tcl Console by selecting **Utility > Tcl Console** (View menu).
3. At a Tcl Console command prompt, type `source <path>/<project name>.tcl` .
4. Open the new project by selecting **Open Project** (File menu) and start compilation by selecting **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 and 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 (LPMs), device-specific embedded megafunctions such as PLLs, LVDS, and digital signal processing (DSP) blocks, intellectual property (IP) available as Altera MegaCore functions, and IP available through the Altera Megafunction Partners Program (AMPP).



Some IP cores may require that you synthesize them in the LeonardoSpectrum software. Refer to the user guide for the specific IP for more information.

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 MegaWizard creates a wrapper file that instantiates the megafunction. The benefits of using the

MegaWizard are that all the parameters are properly set and you do not need any synthesizer library support such as is needed in the direct instantiation method. This is referred to as the black-box methodology.



When directly instantiating megafunctions, see the Quartus II Help to obtain a list of the ports and parameters. Altera recommends using the MegaWizard to ensure that the ports and parameters are set correctly.

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 10–3 shows a summary of the minimum memory sizes and minimum address widths for inferring RAM in various device families.

Table 10–3. Inferring RAM Summary			
	Cyclone II, Stratix GX, Stratix & Cyclone	APEX 20K, APEX 20KE, APEX 20K, 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

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.

Inferring ROM

You can implement ROM behavior in HDL source code with `CASE` statements or specify the ROM as a table. LeonardoSpectrum 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 `lpm_mult` megafunction implements the DSP block in the simple multiplier mode. The following functionality is supported in this mode:

- The DSP block includes registers for the input and output stages and an intermediate pipeline stage
- Signed and unsigned arithmetic is supported

Multiplier Accumulators

The `altmult_accum` megafunction implements the DSP block in the multiply-accumulator mode. The following functionality is supported in this mode:

- The DSP block includes registers for the input and output stages and an intermediate pipeline stage
- The output registers are required for the accumulator
- The input and pipeline registers are optional
- Signed and unsigned arithmetic is supported



If the design requires input registers to be used as shift registers, use the black-boxing method to instantiate the `altmult_accum` megafunction.

Multiplier Adders

The LeonardoSpectrum software can infer multiplier adders and map them to either the two-multiplier adder mode or the four-multiplier adder mode of the DSP blocks. The LeonardoSpectrum software maps the HDL code to the correct `altmult_add` function.

The following functionality is supported in these modes:

- The DSP block includes registers for the input and output stages and an intermediate pipeline stage
- Signed and unsigned arithmetic is supported, but support for the Verilog HDL “signed” construct is limited

Controlling DSP Block Inference

In devices that include dedicated DSP blocks, multipliers, multiply-accumulators, and multiply-adders can be implemented either in DSP blocks, or in logic. You can control this implementation through attribute settings in the LeonardoSpectrum software.

As shown in Table 10–4, 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 10–4. 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 10–4:

- (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 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 and VHDL codes samples show how to use the `extract_mac` attribute.

Using Module Level Attributes in Verilog HDL Code

```
module mult_add ( dataa, datab, datac, datad, result);
//synthesis attribute mult_add extract_mac FALSE
// Port Declaration
input [15:0] dataa;
input [15:0] datab;
input [15:0] datac;
input [15:0] datad;

output [32:0] result;

// Wire Declaration
wire [31:0] mult0_result;
wire [31:0] mult1_result;

// Implementation
// Each of these can go into one of the 4 mults in a
// DSP block
assign mult0_result = dataa * `signed datab;
//synthesis attribute mult0_result preserve_signal TRUE

assign mult1_result = datac * datad;

// This adder can go into the one-level adder in a DSP
// block
assign result = (mult0_result + mult1_result);

endmodule
```

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 only works with signals or wires; it does not work with registers.

Table 10–5 shows the acceptable values for the `dedicated_mult` attribute.

Table 10–5. Values for the <code>dedicated_mult</code> 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 10–5:

- (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 should be 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 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 get 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 for 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 DOWNT0 0);
      m,n,o,p: OUT
      std_logic_vector (35 DOWNT0 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 different control signals for the DSP block, such as sign A, sign B, and dynamic addnsub, use the black-boxing technique.

- While performing signed operations, ensure that the specified data width of the output port matches the data width of the expected result. Otherwise the sign bit may be lost or data may be incorrect because the sign is not extended. For example, if the data widths of input A and B are `width_a` and `width_b`, respectively, then the maximum data width of the result can be $(width_a + width_b + 2)$ for the four-multipliers adder mode. Thus, the data width of the output port should be less than or equal to $(width_a + width_b + 2)$.
- While using the accumulator, the data width of the output port should be equal to or greater than $(width_a + width_b)$. The maximum width of the accumulator can be $(width_a + width_b + 16)$. Accumulators wider than this are implemented in logic.
- If the design uses more multipliers than are available in a particular device, you might 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.



For more information on LogicLock regions and the LogicLock design flow, see the *LogicLock Design Methodology* chapter in Volume 1 of the *Quartus II Handbook*.

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 only apply the LogicLock option in LeonardoSpectrum 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 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 detailed steps:

1. From the Tools menu, switch to the **Advanced Flow** tab instead of the **Quick Setup** tab.
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 tool command language (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 may 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 using the LogicLock import function.

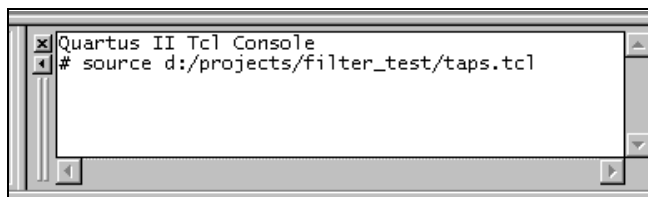
or

- 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 take their EDIF file and create a separate project at that time. New assignments would then have to be added to the top-level project through 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 10-3](#).

Figure 10-3. Tcl Console Window with Source Command



4. Open the new completed project by choosing **Open Project** (File menu), browsing to the project name, and clicking **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 I/O elements (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 sections below describe examples of black-boxing modules using the files described in Figure 1-1 of the *Hierarchical Block-Based & Team Based Design Flows* chapter in Volume 1 of the *Quartus II Handbook*. 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 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](#)” below 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 `nootp 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 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 LeonardoSpectrum software creates a Tcl file for each EDIF file, providing 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. Type `source <path>/<project name>.tcl` 
4. Open the new project by choosing **Open Project** (File menu), browsing to the project name, and clicking **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 (this methodology is not documented here). 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 re-optimize 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 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 re-generated by the LeonardoSpectrum software. They are modules that had a LogicLock assignment in the original compile.

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 Altera technology (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
    dont_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 10–29, you can compile your design using the script.

You can run the script in batch mode at the command prompt using the following command:

```
spectrum -file <Tcl_file> ↵
```

You can also run the script from the interface by choosing **Run Script** (File menu), then browsing to your Tcl file and clicking **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 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.

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 Users Manual*.

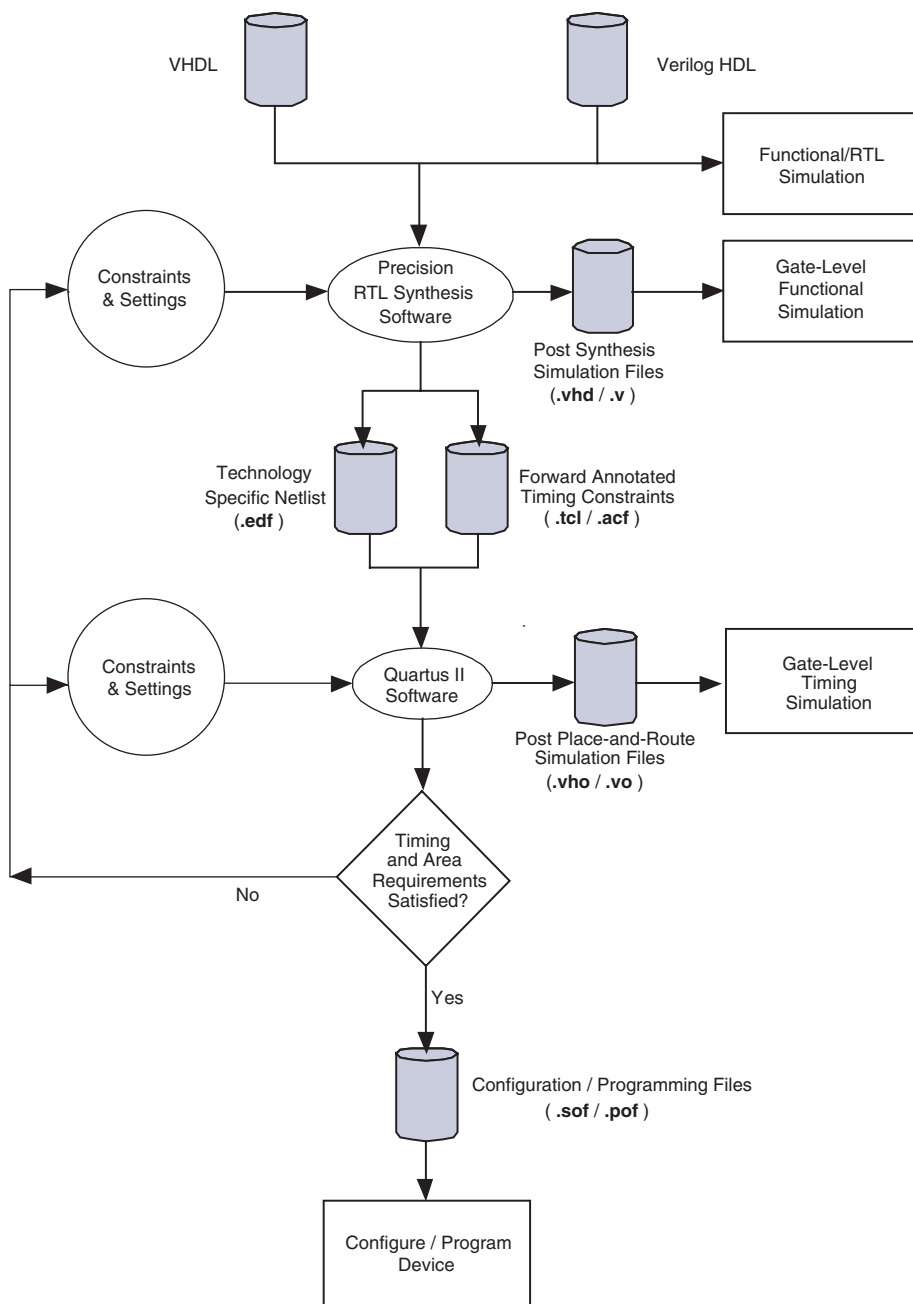
Design Flow

The basic steps in a Quartus II design flow using the Precision RTL Synthesis software are as follows:

1. Create Verilog HDL and/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, select your target device, and set 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 11–1](#) shows the design flow described in the steps above.

Figure 11–1. Recommended Design Flow

As shown in [Figure 11–1](#), 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 re-run 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 11–1](#) lists those files with a short description of each file type.

Table 11–1. 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
.v/.vhd	Post synthesis output design file in Verilog HDL/VHDL format for post synthesis simulation
.rep (1)	Synthesis area and timing report files
.edf	Technology-specific netlist in electronic design interchange format (EDIF)
.acf/.tcl (2)	Forward-annotated constraints file containing constraints and assignments

Notes to Table 11–1:

- (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 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 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 a 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 technology 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 **Additional Output Netlist** (Tools > Set Options > Output 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, 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 common text editor or use the Precision RTL Synthesis software to automatically generate one for you 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 the GUI in a file named `<design_name>.sdc` 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.



See the *Attributes* chapter in the *Precision Synthesis Reference Manual* for details and examples.

Setting Timing Constraints

Timing constraints, based on the industry standard SDC format, are important pieces of information that the Precision RTL Synthesis software needs 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 a 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 for SDC commands, see the *Precision RTL Synthesis Users Manual* and the *Precision Synthesis Reference Manual*. See the *Attributes* chapter in the *Precision Synthesis Reference Manual* available on the Mentor Graphics web site at www.mentor.com for details and examples.

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 SLEW -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 **Set Input Constraints** or **Set Output Constraints** option under **Inputs** or **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 **SLEW** 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, in the Design Hierarchy pane, expand the **Ports** entry.
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 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 pin and I/O registers if 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 Any 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 they 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 un-routable. 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.

To eliminate routability and timing issues associated with high fan-out nets, the Precision RTL Synthesis software also allows you to override the library default value on a global or individual-net basis. You can override the library value by setting a `max_fanout` attribute on the net.

Synthesizing the Design & Evaluating the Results

To synthesize the design for the target device, click on the **Synthesize** icon in the Precision RTL Synthesis Design Bar. During synthesis, the Precision RTL Synthesis software optimizes the compiled design, then writes out netlists and reports to the implementation subdirectory of your working directory after the implementation is saved, using the `<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 Users Manual* 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 as specified in the Precision software, and makes timing assignments. For certain devices to be compiled in the Quartus II software version 4.1 and above, 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 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 tree 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 (i.e. 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 above support the individual execution of various Quartus II executables, such as analysis & 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 above. By default, Precision generates a modular Quartus II Project

Configuration File (Tcl file) for Stratix II, Stratix, Stratix GX, MAX II, and Cyclone 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 families, Precision uses the "Quartus II" flow option, which enables the Quartus II compilation flow that existed in Precision versions prior to 2004a, and was supported in Quartus II versions prior to 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 above.

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 & 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 the Quartus II software versions prior to 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, only allows the "MAX+PLUS II" flow.

After the design is compiled in the Quartus II software from within the Precision RTL Synthesis software, you can also 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 decide 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 on 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 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-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 timing estimates and take better advantage of timing driven optimization.

This clear box feature of the MegaWizard 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 this option on will cause the Quartus II MegaWizard 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 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 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-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 thus not taking full advantage of the synthesis tool's timing driven optimization.

Using MegaWizard-generated Verilog HDL Files for Black Box Megafunction Instantiation

The MegaWizard 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 in 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-generated VHDL Files for Black-Box Megafunction Instantiation

The MegaWizard 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 in 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 `lpm_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

By default, the Precision RTL Synthesis software sets the `lpm_mult` parameter called `DEDICATED_MULTIPLIER_CIRCUITRY` to `AUTO` to allow the Quartus II software the flexibility to choose regular logic (LEs or ALMs), DSP blocks, or dedicated multiplier logic depending on the device utilization and the size of the multiplier.

If the number of multipliers in your design exceeds the number of dedicated resources in the selected device, you can use the Precision RTL Synthesis GUI or HDL attributes to redirect the mapping of specific operators to logic elements. The options for multiplier mapping in the Precision RTL Synthesis Software are shown in [Table 11-2](#)

Table 11-2. Options for `DEDICATED_MULT` Parameter to Control Multiplier Implementation (Part 1 of 2)

Value	Description
ON	<code>lpm_mult</code> inferred and multipliers implemented in DSP blocks
OFF	<code>lpm_mult</code> inferred, but multipliers implemented in logic by the Quartus II software

Table 11–2. Options for DEDICATED_MULT Parameter to Control Multiplier Implementation (Part 2 of 2)

Value	Description
LCCELL	lpm_mult not inferred and multipliers implemented in logic by the Precision RTL Synthesis software
AUTO	lpm_mult inferred, but multipliers implemented in either logic or DSP blocks by the Quartus II software based on the device utilization and the size of the multiplier

Using the GUI

Take 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 11–2](#):

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 only be applied 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 unsignedmult (result, a, b);
    output [15:0] result;
    input [7:0] a;
    input [7:0] b;
    assign out = 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 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 `altmult_add` or `altmult_accum` megafunction as appropriate for your design. These megafunctions allow the Quartus II software the flexibility to choose regular logic or DSP blocks depending on the device utilization and the size of the function.

You can use the `extract_mac` attribute to prevent the inference of an `altmult_add` or `altmult_accum` megafunction in a certain module or entity. The options for this attribute are shown in [Table 11–3](#).

<i>Table 11–3. Options for EXTRACT_MAC Attribute Controlling DSP Implementation</i>	
Value	Description
TRUE	The <code>altmult_add</code> or <code>altmult_accum</code> megafunction is inferred
FALSE	The <code>altmult_add</code> or <code>altmult_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 11–3](#).

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_altmult_accum1 (dataout, dataa, datab, clk, aclr, clken);
    input [7:0] dataa, datab;
    input clk, aclr, clken;

    output [31:0] dataout;
    reg [31:0] dataout;

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

    assign multa = dataa * datab;

    //synthesis attribute multa preserve_signal TRUE
    //synthesis attribute multa dedicated_mult OFF
    assign adder_out = multa + dataout;

    always @ (posedge clk or posedge aclr)
    begin
        if (aclr)
            dataout <= 0;
        else if (clken)
            dataout <= adder_out;
        end

    //synthesis attribute unsig_altmult_accum1 extract_mac FALSE
endmodule
```

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 where 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 the 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*. For more information on hierarchical design methodologies and design flows using the Quartus II software, see the *Hierarchical Block-Based and Team-Based Design Flow* chapter in Volume 1 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 Precision RTL Synthesis projects for each lower-level block.

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

Use 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 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 Tcl file contains the following commands for each LogicLock region:

```
project add_assignment {filter} {taps_region} {} {} {LL_AUTO_SIZE} {ON}
project add_assignment {filter} {taps_region} {} {} {LL_STATE} {FLOATING}
project add_assignment {filter} {taps_region} {} {taps:ul} {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 once.

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 where 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 the lower-level modules and when setting up the top-level design, follow these guidelines.

For lower-level modules:

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.

For top-level designs:

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 using the files described in the *Hierarchical Block-Based & Team-Based Design Flows*, chapter in Volume 1 of the *Quartus II Handbook*. 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, 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.

Example:

```
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 allows you to control how your design files will be prepared for the Quartus II place-and-route process and 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*.



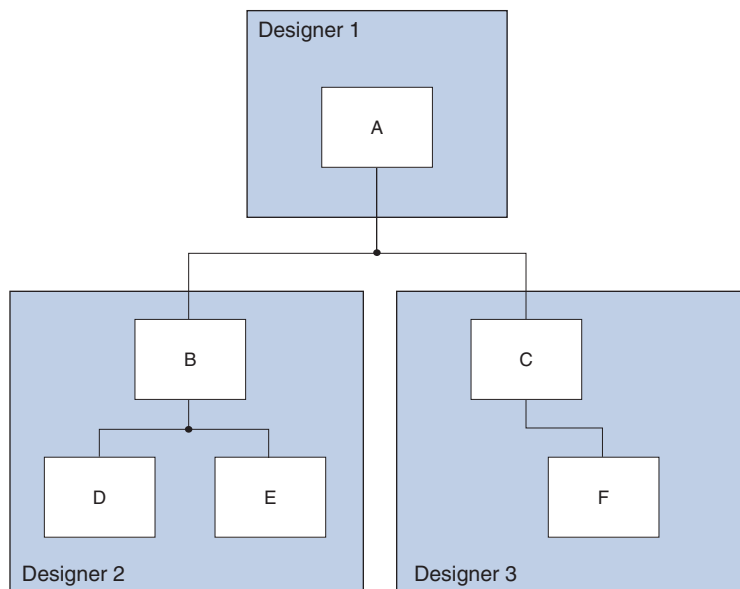
This chapter assumes that you have set-up and licensed, and are familiar with the FPGA Compiler II software.



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 12–1](#) shows an example of a design hierarchy.

Figure 12–1. Design Hierarchy for Block-Based Designs

In [Figure 12–1](#), 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 12–2](#) shows a block design hierarchy.

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 12–1](#) summarizes the structure of [Figure 12–2](#).

Table 12–1. Synthesis in Block-Level Methodology			
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 synthesized 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 12–2](#) 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 will not be 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.

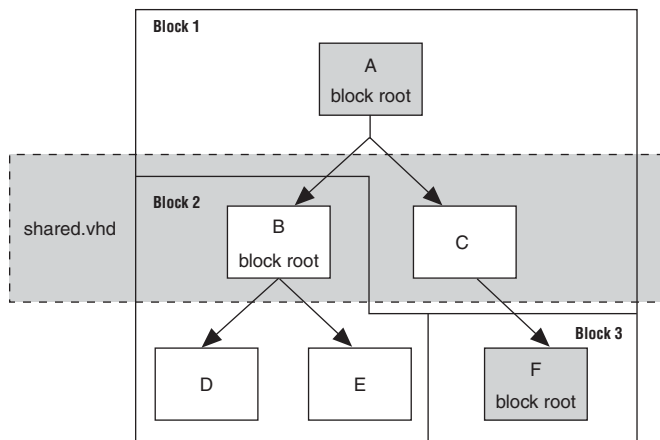
Hierarchy Considerations

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 12-3.

Figure 12-3. Shared Source File Causes Re-Synthesis of Multiple Blocks



In Figure 12-3, 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 12-1, 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 12-3](#), 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 had previously verified B and later made changes to C, then B will be 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.

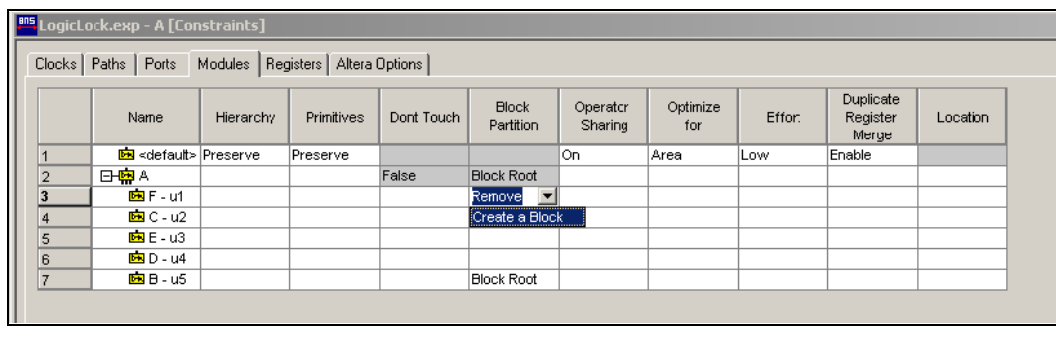
Opening the Modules Constraint Table & Labeling Block Roots

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 12–4 shows how to label block roots.

Figure 12–4. Labeling Block Roots in the Edit Constraints Window



Exporting Block-Level Netlist Files

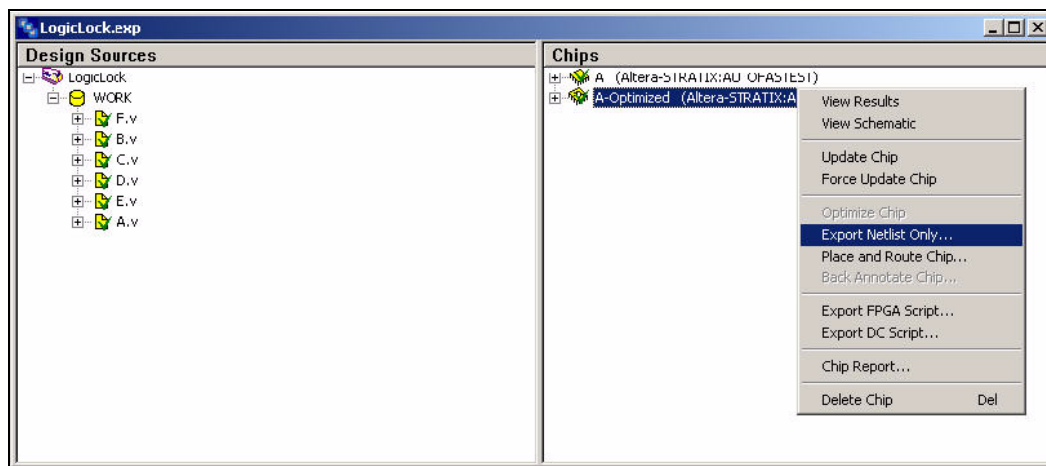
Once your design has been segmented into blocks and re-implemented, you can export netlist files. To export netlist files, perform the following steps:

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 12–8](#). Right-click on an optimized implementation and select **Export Netlist Only** (see [Figure 12–5](#)).

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 12–5. 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**.

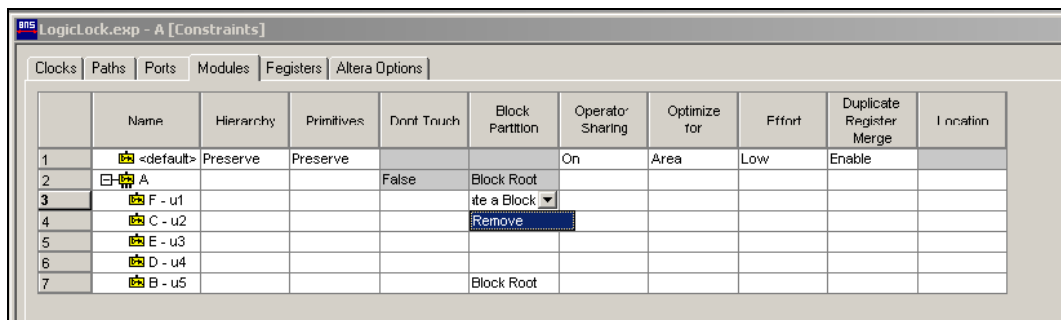
When you update all of the parts of the design, new netlist files will be created for only those parts that have been changed. You can check the time stamps of the new files to confirm this.

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 12-6](#).

Figure 12-6. Removing a 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 hierarchical 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.

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:
www.synopsys.com/products/dcfpga/dcfpga.html

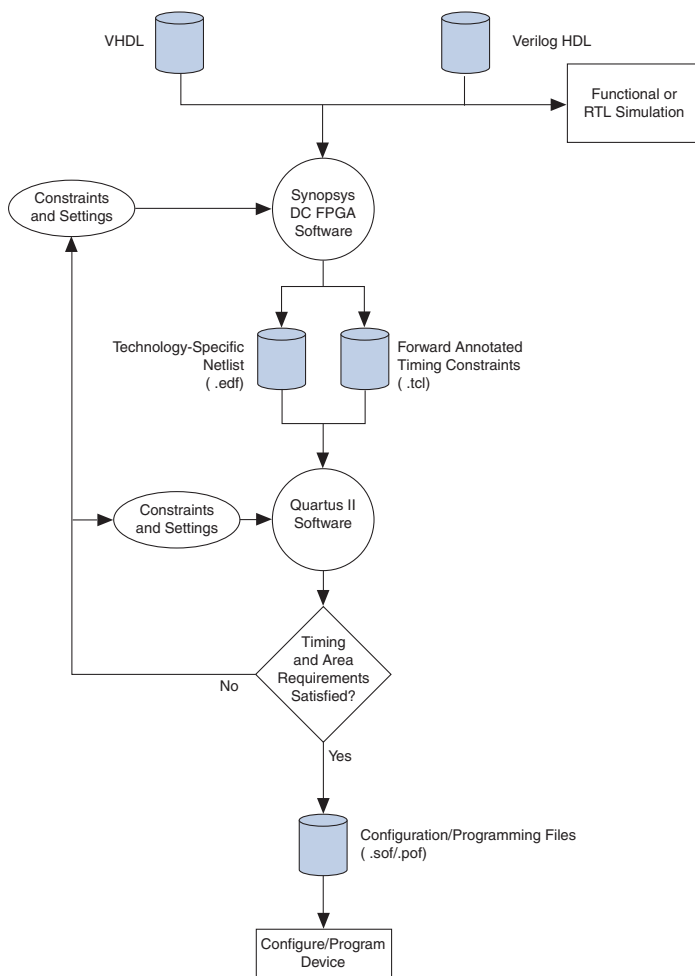
The following areas are covered in this chapter:

- 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 13–1](#).

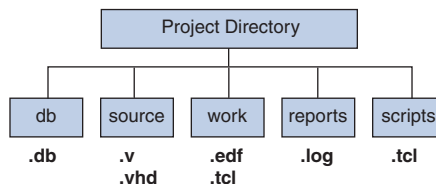
Figure 13–1. Design Flow Using the DC FPGA Software & the Quartus II Software



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 13–2.

Figure 13–2. 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 will automatically read 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 in the

`<installation_path>/dc_fpga/2004.06/libraries/2004.06/` 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 "<installation_path>/dc_fpga/2004.06/libraries/2004.06"

set search_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"

define_design_lib altera_mf -path $dcfpga_lib_path/STRATIX/altera_mf_lib
define_design_lib LPM -path $dcfpga_lib_path/STRATIX/LPM
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 will provide 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 13–7](#).



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.

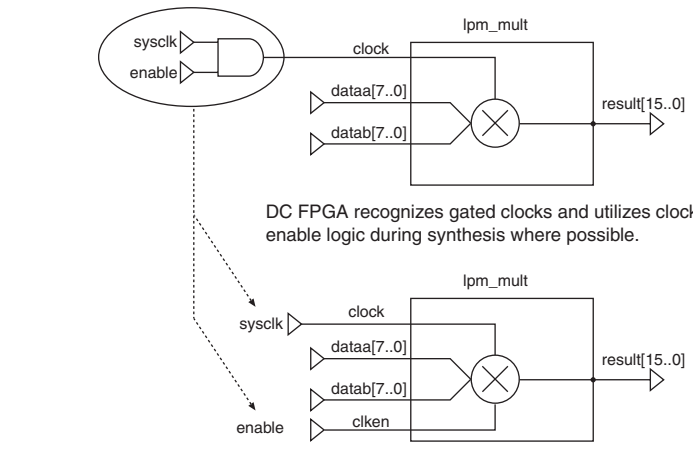
If you instantiate a megafunction in your HDL code, you can use the MegaWizard Plug-In Manager to parameterize the function or you can instantiate the function using the port and parameter definition. The MegaWizard Plug-In Manager provides a graphical interface in the Quartus II software for customizing and parameterizing megafunctions. The section [“Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager” on page 13–7](#) describes the MegaWizard flow with the DC FPGA synthesis software.

There are two ways of instantiating MegaWizard-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.

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 13-3](#).

Figure 13-3. Gated Clock Optimization



The DC FPGA software does not perform gated clock optimization on instantiated black box megafunctions or on instantiated megafunction variation wrapper file. The DC FPGA software performs gated clock optimization on synthesizable inferred megafunctions.

Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager

When you use the MegaWizard to set up and parameterize a megafunction and create a custom megafunction variation, the MegaWizard creates either a VHDL or Verilog HDL wrapper file that instantiates the megafunction.

Reading MegaWizard-Generated Variation Wrapper Files

The DC FPGA software has the ability to analyze and elaborate the **MegaWizard-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 takes advantage of this variation wrapper file during the optimization of your design to reduce area utilization and improve path delays.

Using the megafunction variation wrapper file *<output file>.v* or *<output file>.vhd* in the DC FPGA software synthesis provides good synthesis results for area estimates, but actual timing results are best predicted after place and route inside the Quartus II software. However, reading the megafunction variation wrapper allows the DC FPGA software to provide better synthesis estimates over a black-box methodology.

Using MegaWizard-Generated Variation Wrapper Files in a Black-Box Methodology

Instantiating the MegaWizard-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 MegaWizard-Generated Verilog HDL Files for Black-Box Megafunction Instantiation

By default, the MegaWizard 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 MegaWizard-Generated VHDL Files for Black-Box Megafunction Instantiation

By default, the MegaWizard 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.

Inferring Altera Megafunctions from HDL Code



The DC FPGA synthesis engine 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 will cause your logic to consume more area in your device and may adversely affect your design performance. Altera logic 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, in a Stratix device there are dedicated DSP blocks, but a Cyclone™ device does not have them. In a Cyclone device, multipliers are implemented in LEs.

An example of Verilog HDL code that infers a two-port RAM that can be synthesized into an M512 RAM block of a Stratix device is shown below:

```
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

```

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 need to identify which of the acceptable languages the file is written in when reading designs into the DC FPGA software. The acceptable HDL languages are listed in [Table 13–1](#).

Table 13–1. Supported Design File Formats

Format	Description	Keyword	Extension
Verilog (Synopsys Presto HDL)	Verilog hardware description language	verilog	.v
VHDL	VHSIC hardware description language	vhdl	.vhd
.db	Synopsys internal database format (1)	db	.db
EDIF	Electronic design interchange format	edif	.edf

Note to [Table 13–1](#):

(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 will perform 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 needs to 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 MegaWizard-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 below:

Selecting a Target Device

```
fpga_shell-t> set_fpga_target_device -show_all
Loading db file
'/dc_fpga/2004.06/libraries/2004.06/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 wanted to target the
C6_HARDCOPY_FPGA_PROTOTYPE of the EP1S25F672 Stratix device,
you would 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 (SDC) are Tcl-format commands that are widely used in many EDA software applications. The DC FPGA software accepts all the SDC commands that the full version of the Design Compiler software uses. However, certain constraints that are used in ASIC synthesis are not applicable to programmable logic synthesis, so the DC FPGA software ignores them.

The accepted constraints for use in the DC FPGA software are:

- `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_pad_type`
- `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*.



Minimum timing analysis is not necessary for synthesis with the DC FPGA software because the software 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.

The timing reports generated from the DC FPGA software are preliminary estimates of the path delays in your design, and accurate timing will only come 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 13-2](#).

Table 13-2. Reporting Commands

Object	Command	Description
Design	report_design	Reports design characteristics
	report_area	Reports design size and object counts
	report_hierarchy	Reports design hierarchy
	report_resources	Reports resource implementations
	report_fpga	Reports FPGA resource utilization statistics for the design
Instances	report_cell	Displays information about instances
References	report_reference	Displays information about references
Ports	report_port	Displays information about ports
	report_bus	Displays information about bused ports
Nets	report_net	Reports net characteristics
	report_bus	Reports bused net characteristics
Clocks	report_clock	Displays information about clocks
Timing	report_timing	Checks the timing of the design
	report_constraint	Checks the design constraints
	check_timing	Checks for unconstrained timing paths and clock-gating logic
	report_design	Shows operating conditions, timing ranges, internal input and output, and disabled timing arcs.
	report_port	Shows unconstrained input and output ports and port loading
	report_timing_requirements	Shows all timing exceptions set on the design
	report_clock	Checks the clock definition and clock skew information
	derive_clocks	Checks internal clock and unused registers
	report_path_group	Shows all timing path groups in the design
Cell Attributes	get_cells	Shows all cell instances that have a specific attribute



For more information 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 path delays because the timing of your design cannot be accurately predicted until a full place and route of the design has been performed in the Quartus II software.

Saving Synthesis Results

After synthesis, the technology-mapped design can be saved to a file in four formats: Verilog HDL, VHDL, Synopsys internal DB, or EDIF.

Currently, 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:

```
write -format <verilog|vhdl|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 for when you have a design synthesized and want to reuse it later in the DC FPGA software. The DB file contains your constraints and synthesized design netlist, and loads into DC FPGA faster than Verilog HDL or VHDL designs.

You can also write out your applied design constraints in Tcl format for export to the Quartus II software environment. You can do this with the `write_par_constraint` command, which 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 from the DC FPGA software, 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 in 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 so that you can compile your design immediately.

For example, assuming your current design in the DC FPGA software is **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 software shell, run this script 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 ←
```

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 used.

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 2004.03 on Wed Apr 28 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 for your EDIF netlist. This 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 all 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 are ready to perform the place and route tasks in the Quartus II software. The Synopsys DC FPGA software works only on worst case timing delay and constraints, and does not work to optimize minimum timing requirements. Altera recommends adding minimum timing constraints and performing minimum timing analysis inside the Quartus II software.

Conclusion

Large PLD 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.



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

qii51013-2.0

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 your debugging, optimization, or constraint entry process.

The first sections of this chapter explain the different parts of the RTL Viewer and Technology Map Viewer. The following sections describe how to navigate and filter in the schematics, probe to other features within the Quartus II software, view a timing path from the Timing Analyzer report and other features of the schematics, and how you can navigate through your design netlist. The final section [“Using the RTL & Technology Map Viewers to Analyze Design Problems” on page 14–28](#) 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), and/or schematic (.bdf or .gdf imported from the MAX+PLUS® II software) design files. 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 has 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 netlist. 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 viewers group 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. You can then run the viewer by choosing **RTL Viewer** (Tools menu), or by selecting **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.

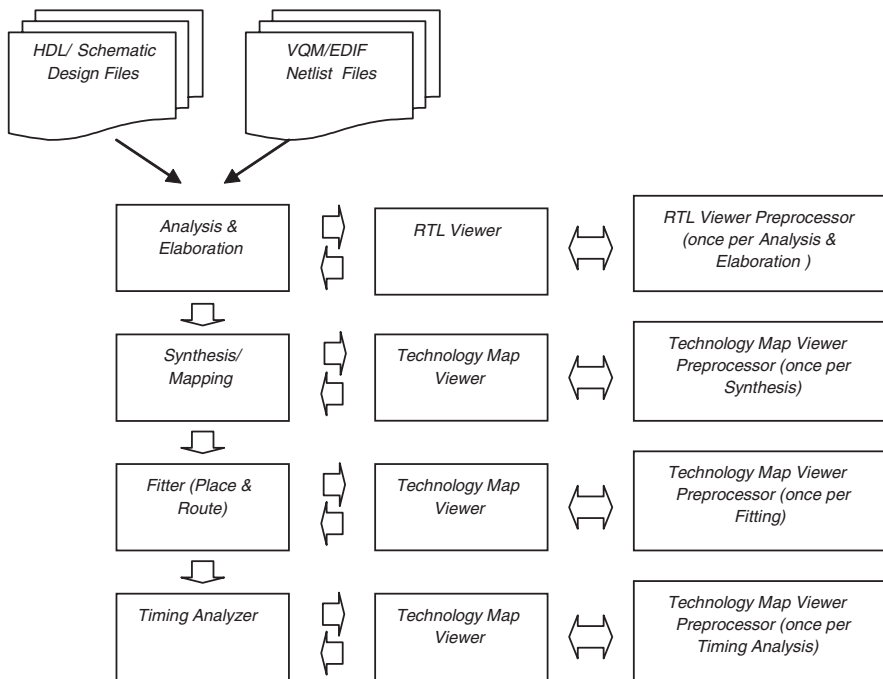
You can view your Quartus II technology-mapped results after synthesis, fitting, or timing analysis. To run the Technology Map Viewer for a Quartus II project, you must first synthesize and map the design to the target technology by choosing **Start > Start Analysis & Synthesis** (Processing menu). You 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 14–22 for details).

Once 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 14–1 shows how the RTL Viewer and Technology Map Viewer fit into the basic Quartus II design flow.

Figure 14–1. 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 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.

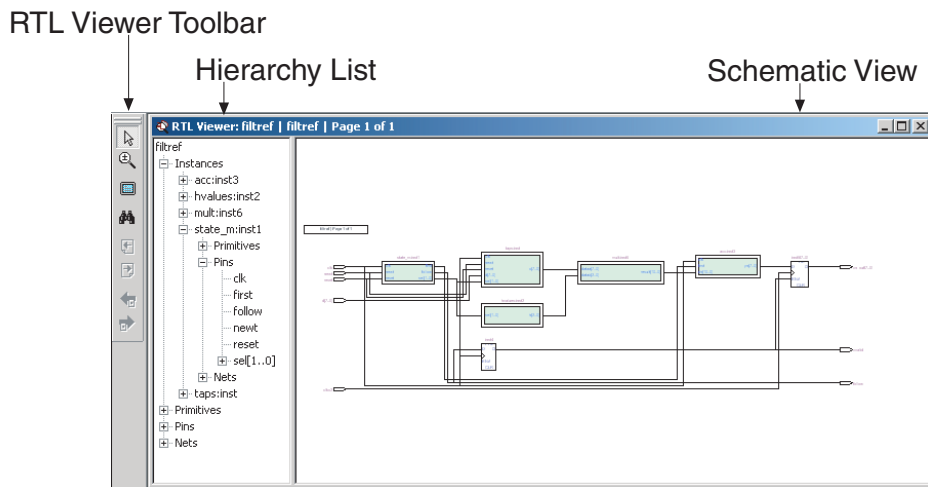
If the viewer window is open when you start a new compilation, the viewer closes automatically. You must open the viewer again to view the new design netlist.

Introduction to the User Interface

The RTL Viewer window and Technology Map Viewer window each consist of two main parts, as shown in [Figure 14–2](#) 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 like 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 14–2. RTL Viewer Window & RTL Viewer Toolbar



Schematic View

The schematic view is displayed on the right-hand side of the RTL Viewer or 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.

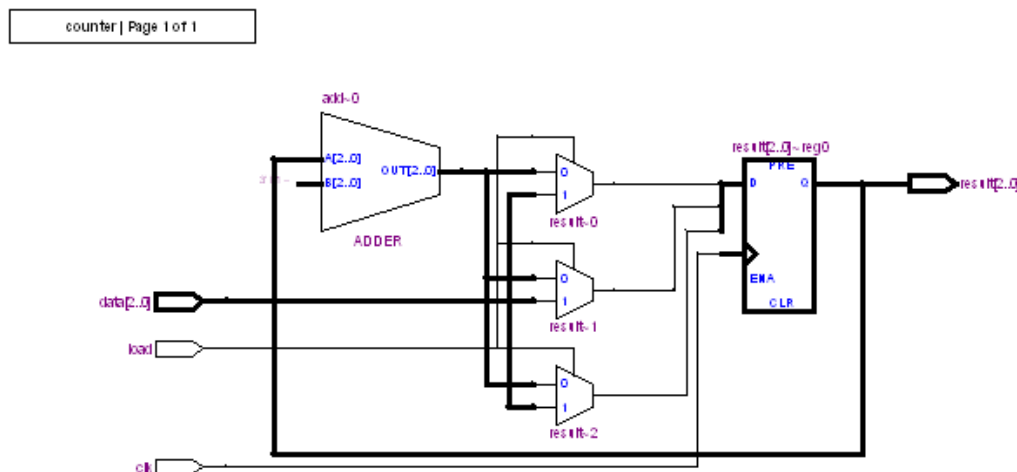
Schematic Symbols

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 14–3 shows an example of an RTL Viewer schematic for a 3-bit synchronous loadable counter. The code sample in the “Code Sample for Counter Schematic Shown in Figure 14–3” 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 14–1) along with an ADDER operator (see Table 14–2) 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 14–3. Example Schematic Diagram in the RTL Viewer



Code Sample for Counter Schematic Shown in Figure 14–3

```

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 14–4 shows a portion of the corresponding Technology Map Viewer schematic. 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 hexadecimal number in parentheses below each LCELL primitive represents the LUT Mask, which is a hexadecimal representation of the look-up table (LUT) output.

Figure 14–4. Example Schematic Diagram in the Technology Map Viewer

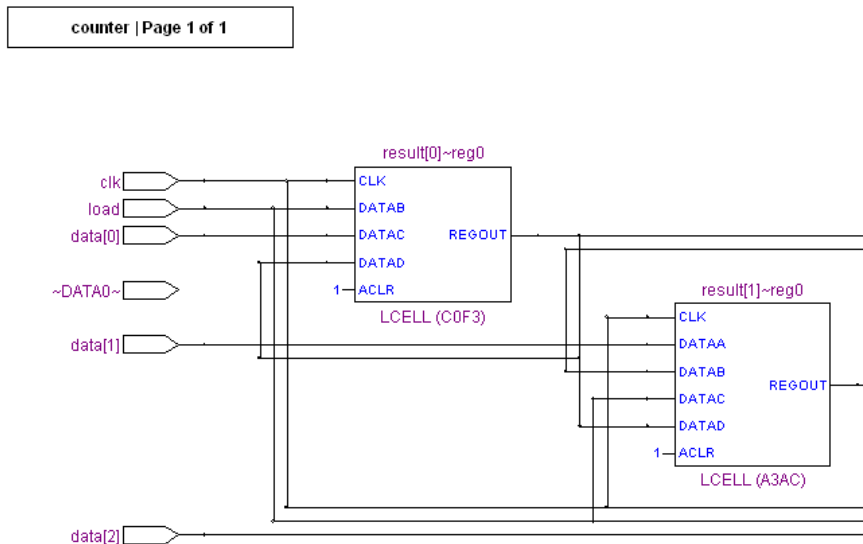


Table 14–1 lists and describes the primitives and basic symbols that can be displayed in the schematic view of the RTL Viewer and Technology Map Viewer. Note that the logic gates and operator primitives only appear in the RTL Viewer, while the logic in the Technology Map Viewer is represented by atom primitives such as LCELL (logic cell).

Table 14–1. Symbols in the Schematic View (Part 1 of 3)

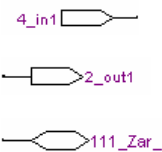
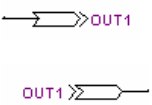

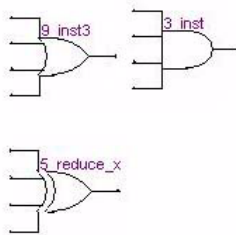
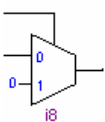
Symbol	Description
<p>I/O Ports</p> 	<p>An input, output, or bidirectional port in the current level of hierarchy. A device input, output, or bidirectional pin when viewing the top-level hierarchy. The symbol can represent a bus. Only one wire is connected the bidirectional symbol, representing both 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.</p>
<p>I/O Connectors</p> 	<p>An input or output connector, representing a net that comes from another page of the same hierarchy (see “Page Partitioning in the Schematic View” on page 14–14). 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 14–15).</p>
<p>Hierarchy Port Connect</p> 	<p>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.</p>
<p>OR, AND, XOR Gates</p> 	<p>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.</p>
<p>MUX</p> 	<p>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 14–9).</p>

Table 14–1. Symbols in the Schematic View (Part 2 of 3)

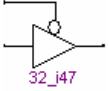
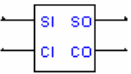
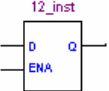
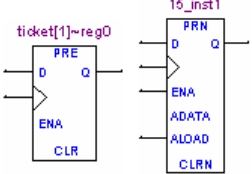
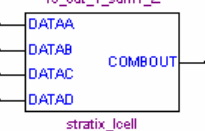
Symbol	Description
<p>BUFFER</p> 	<p>A buffer primitive. The figure shows the tri-state buffer, with an inverted output enable port. Other buffers without an enable port include <code>LCELL</code>, <code>SOFT</code>, <code>CARRY</code>, and <code>GLOBAL</code>. The <code>NOT</code> gate and <code>EXP</code> expander buffers use this symbol without an enable port and with an inverted output port.</p>
<p>CARRY_SUM</p> 	<p>A <code>CARRY_SUM</code> buffer primitive, where <code>SI</code> represents SUM IN, <code>SO</code> represents SUM OUT, <code>CI</code> represents CARRY IN and <code>CO</code> represents the CARRY OUT port of the buffer.</p>
<p>LATCH</p> 	<p>A latch primitive with <code>D</code> data input, <code>ENA</code> enable input, and <code>Q</code> data output.</p>
<p>DFFE/ DFFEA</p> 	<p>A <code>DFFE</code> (data flip-flop with enable) primitive, with the same ports as a latch and a clock trigger, or a <code>DFFEA</code> (data flip-flop with enable and asynchronous load) primitive, with additional asynchronous load and data signals. Preset and Clear connections are also shown if these ports are connected.</p>
<p>Other Primitive</p> 	<p>Any primitive that does not fall into the categories above. These 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.</p> <p>The figure shows a <code>stratix_lcell</code> WYSIWYG primitive, with <code>DATAA</code> to <code>DATAD</code> and <code>COMBOUT</code> port connections. This type of <code>LCELL</code> 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.</p>

Table 14–1. Symbols in the Schematic View (Part 3 of 3)

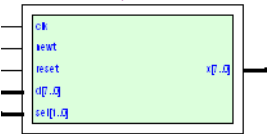
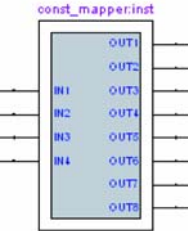
Symbol	Description
<p>Instance</p> 	<p>An instance in the design that does not correspond to a primitive or operator (generally a user-defined hierarchy block), indicated by the double outline and green coloring. The symbol displays the instance name. You can open the schematic for the lower-level hierarchy by right-clicking and choosing the appropriate command (see “Traversing the Design Hierarchy” on page 14–16).</p>
<p>Encrypted Instance</p> 	<p>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.</p>

Table 14–2 lists and describes the additional higher-level operator symbols used in the RTL Viewer schematic view.

Table 14–2. Operator Symbols in the Schematic View (Part 1 of 3)

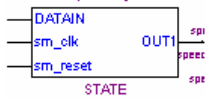
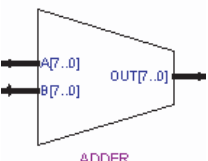
Symbol	Description
<p>speed:legal</p> 	<p>A particular state of an finite state machine, with the following ports:</p> <ul style="list-style-type: none"> DATAIN - input data that control the state OUT1 - output of that state sm_clk - Clock input feeding the state sm_reset - Reset input feeding the state sm_enable - Clock Enable signal feeding the state
<p>add~0</p> 	<p>An adder operator: OUT = A + B</p>

Table 14–2. Operator Symbols in the Schematic View (Part 2 of 3)

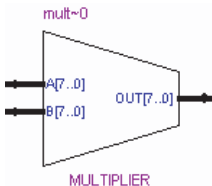
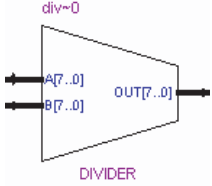
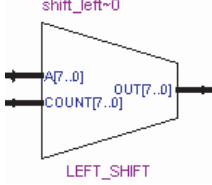
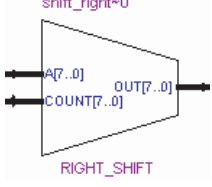
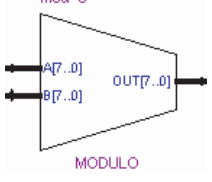
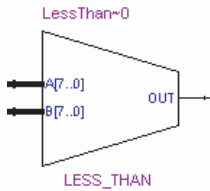
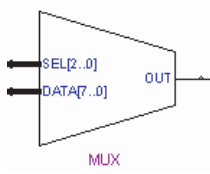
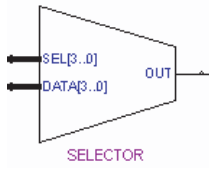
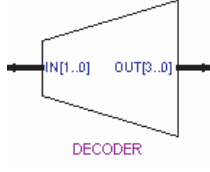
Symbol	Description
	<p>A multiplier operator: $OUT = A \times B$</p>
	<p>A divider operator: $OUT = A / B$</p>
	<p>A left shift operator: $OUT = \{A \ll COUNT\}$</p>
	<p>A right shift operator: $OUT = \{A \gg COUNT\}$</p>
	<p>A modulo operator: $OUT = (A \% B)$</p>

Table 14–2. Operator Symbols in the Schematic View (Part 3 of 3)

Symbol	Description
 <p>LessThan~0</p> <p>LESS_THAN</p>	<p>A less than comparator: $OUT = \{A \leq B : A < B\}$</p>
 <p>Mux~0</p> <p>MUX</p>	<p>A multiplexer: $OUT = DATA[SEL]$ The data range size is $2^{\text{sel range size}}$</p>
 <p>Select~0</p> <p>SELECTOR</p>	<p>A multiplexer with one-hot select input (and more than two input signals).</p>
 <p>Decoder~1</p> <p>DECODER</p>	<p>A binary number decoder: $OUT = (binary_number(IN) == x)$ for $x=0$ to $x=2^{(n+1)} - 1$</p>

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) 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 to be selected, 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 will highlight 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 choose **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](#)” section). 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 may be connected on other pages of the design, but these nets are still highlighted on the current page 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 page. Right-click in the schematic and choose **Viewer Options**. In the **Net Selection** section, turn on the **Limit selections to current page** 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 when using Quartus II integrated synthesis, or logic cell atoms from a VQM or EDIF when using 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 names in the hierarchy list, the RTL Viewer performs the following actions:

1. Displays the hierarchy and page that contain the selected item in the schematic view if it is not currently displayed.
2. Centers the current schematic page to include the selected item, if needed.
3. 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

This section describes various ways of navigating through the pages and hierarchy levels in 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 choosing **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 schematics may be displayed at the default normal size. Choose **Zoom In** to see less of the image in a larger size, and choose **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 choose **Fit Selection in Window** to enlarge the area covered by those selection nodes. 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 enable 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 enabled, you may 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 will be 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 nothing in the schematic is selected. If something in the view is selected, the zoom centers the view on the selected items.

Page Partitioning in the Schematic View

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 1000. The **Ports per page** option specifies the number of ports (or pins) per partitioned page. The default value is 1000 ports, and the range is 1 to 2000. 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 viewer window (or for the Quartus II software when the viewer window is maximized) 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 14–5 shows an example.

Figure 14–5. RTL Viewer Title Bar Indicating Page Number Information



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.

You can go to a particular page of the schematic by choosing **Go To** (Edit menu), or by right-clicking in the schematic view, then choosing the **Go To** command and selecting the page number in the **Page** list.

Following Nets Between Schematic Pages

Input connectors and output connectors are used to represent nodes that connect between pages of the same hierarchy. Right-clicking on a connector provides 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 and 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 to highlight in red before you right-click to traverse between pages of the hierarchy.

Input Connectors

Figure 14–6 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.

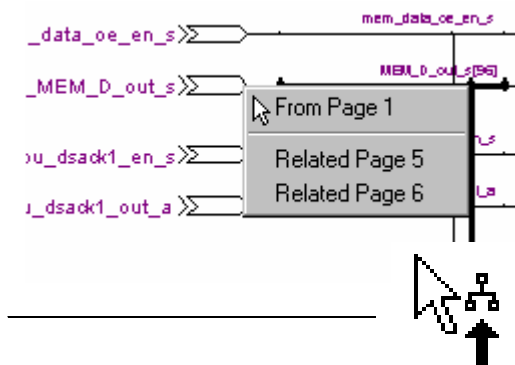
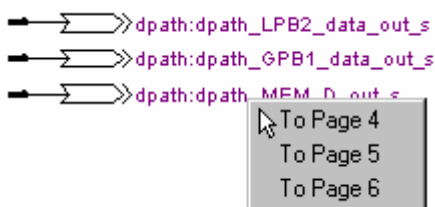
Figure 14–6. Input Connector Right-Click Menu**Output Connectors**

Figure 14–7 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 14–7. Output Connector Right-Click 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 14–12), or by using the **Hierarchy Up** and **Hierarchy Down** commands in the schematic view described in this section.

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 enabled, the appropriate option is available when your mouse cursor is located over an area of the schematic view that has a corresponding lower- or higher-level hierarchy.

The mouse pointer changes as it hovers over different areas of the schematic to indicate from where in the schematic you can move down or up or both in the hierarchy (as shown in Figure 14–8). To open the next hierarchy level, right-click in that area of the schematic and select **Hierarchy Down** or **Hierarchy Up**, as appropriate, or double-click in that area of the schematic.

Figure 14–8. Mouse Pointers Indicating that Hierarchy Down, Hierarchy Up, or Both Down and Up Are Available



Back & Forward Page Viewing

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 intended only for switching between page views; they cannot be used to undo an action such as a selection of 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 **Go to Net Driver**. 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 related to one or more specific nodes or nets.

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 choose 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 “[Back & Forward Page Viewing](#)”).



When viewing a filtered netlist, if you click an item in the hierarchy list, the schematic view displays 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**.

- **Sources**—This command filters to 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. If an input port is selected, the filtered page shows only the input source nodes that feed this port. (Note that 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.
- **Destinations**—This command filters to all 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. If an output port is selected, the filtered page shows only the fan-out destination nodes fed by this port. (Note that 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.
- **Sources & Destinations**—This is a combination of the **Sources** filtering command and the **Destinations** filtering command, where the filtered page shows both the sources and the selected node.
- **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.)
- **Selected Nodes & Nets**—This command creates a filtered page that shows only the selected node(s) and, if applicable, the connections between those nodes. If a net is selected, the filtered page shows the immediate sources and destinations of the selected net.

The filtering commands apply to nodes in the same netlist hierarchy by default. The **RTL/Technology Map Viewer Options** dialog box provides an option to filter through levels of hierarchy. Right-click in the schematic and choose **Viewer Options**. In the **Filtering** section, turn on the **Filter across hierarchy** option. When the filtered path passes through a level of hierarchy, a diamond shape symbol appears in the schematic, representing a port relationship between two different hierarchies.

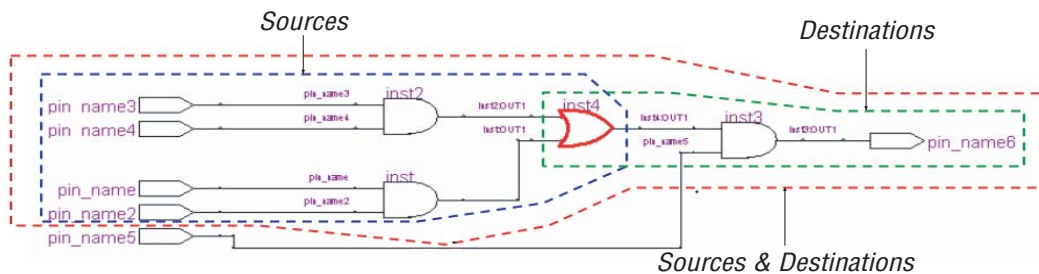
For all the filtering commands, the viewer stops tracing through the netlist to obtain the filtered netlist when it reaches one of the following:

- A pin
- A specified number of filtering levels (counting from the selected node or port; 10 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 10 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 14–9 shows an example of the **Sources**, **Destinations**, and **Sources & Destinations** commands for the `inst4` node highlighted in the schematic. Figure 14–10 shows an example of the **Between Selected Nodes** 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 choose the corresponding command.

Figure 14–9. Example Schematic with “Sources,” “Destinations,” & “Sources & Destinations” Filtering for `inst4`



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 expand 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 choose the **Expand** command. This command expands logic from the selected port by the amount specified in the **Viewer Options**. You can set these options by right-clicking in the schematic view, and choosing **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 10 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 will be shown in the expanded schematic.

The **Expand** 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 14–17](#) 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 turned on by default for 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 Features

The RTL and Technology Map Viewers provide the ability to cross-probe from the viewer to the source design file or various other features 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 features, allowing you to view and make changes or assignments in the appropriate editor or floorplan.

Locate by right-clicking the node (or nodes) of interest in the schematic and choosing the appropriate locate command, **Locate in Assignment Editor**, **Locate in Design File**, **Locate in Timing Closure Floorplan**, **Locate in Last Compilation 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 place and route. 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 place and route 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 editor or floorplan window as needed.

Viewing a Timing Path in the Technology Map Viewer

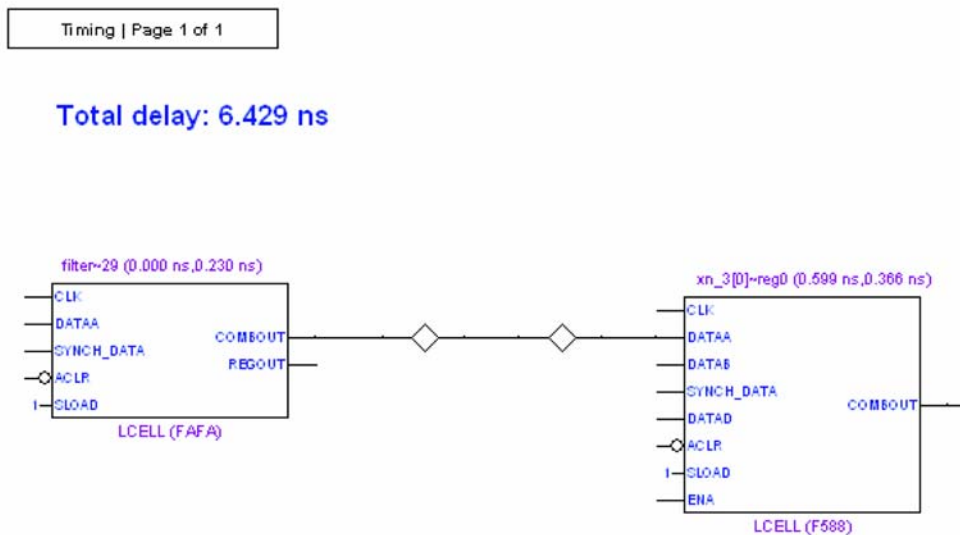
You can view a timing path from the Timing Analyzer in the Technology Map Viewer. This provides a visual representation of the information that can be obtained by listing the paths from the **Timing Analyzer** window.

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 select any of the detailed reports for **Clock Setup**:<clock name>, **tsu**, **tco**, **tpd**, etc. 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 in Technology Map Viewer**. After you choose this command, the Technology Map Viewer window is opened, or brought into the foreground if already open. Note that the first time the window is opened after a compilation, the Preprocessor stage runs before the window can be opened.

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 in the following format: *<post-synthesis node name> (<IC delay> ns, <cell delay> ns)*.

Figure 14–13 shows a portion of a timing path represented in the Technology Map Viewer. The total delay for the entire path that goes through eight levels of logic in this example (only two of which are shown in the figure) is 6.429 ns. The cell delay through the first LCELL primitive is 0.230 ns (there is no interconnect or IC delay for this first node in the path). The second LCELL primitive has an interconnect delay of 0.599 ns representing the path between the first and second LCELL, and a cell delay of 0.366 ns through the second LCELL. When the timing path passes through a level of hierarchy, a diamond shape symbol appears in the schematic, representing a port relationship between two different hierarchies.

Figure 14–13. 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.

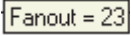
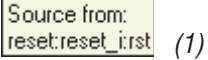
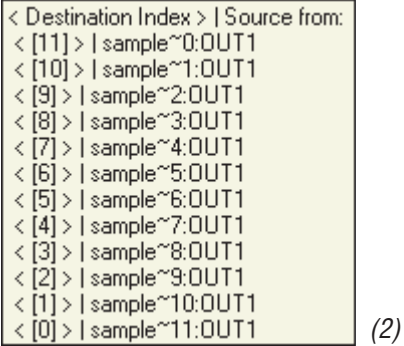
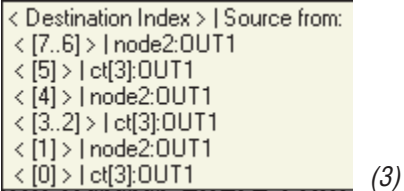
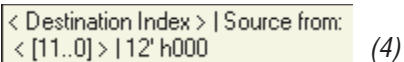
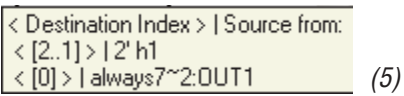
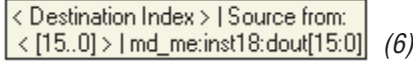
Tooltips

When hovering or holding the mouse pointer over various parts of the schematic, tooltips are displayed. Tooltips contain useful information about nodes (instances, primitives, and pins), nets, input ports and output ports.

Table 14–3 lists the information contained in the tooltips for each type of node:

Table 14–3. Tooltips Information	
Example Tooltips	Description & Tooltip Format
control:controller, INST	Instance Format: <instance name>, <instance type>
clocks:inst7 Mux~1, OPER (MUX) md_me:inst18 data[3..3], DFFE	Primitive Format: <primitive name>, <primitive type>
pc_clock, INPUT Test_probe, OUTPUT	Pin Format: <pin name>, <pin type>
node2_OUT1	Connector Format: <connector name>
md_me:inst18:dout[15..0], Fanout = 2	Net Format: <net name>, Fanout = <number of fanout signals>

Table 14–3. Tooltips Information

Example Tooltips	Description & Tooltip Format
	Output port Format: Fanout = <number of fanout signals>
     	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 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).

You can disable the appearance of tooltips or control the time that the tooltips are displayed in the **Tooltip settings** on the **RTL/Technology Map Viewer** tab of the **Options** dialog box (Tools menu).

The **Show names in tooltip for** option specifies the number of seconds you want to display the names of assigned nodes and pins in a tooltip when you place the pointer over the assigned cells and pins. You can type or select a number. Selecting **Unlimited** causes the tooltip to display indefinitely, as long as the pointer remains over the cell or pin. Selecting **0** turns off tooltips. The default value is 5.

The **Delay showing tooltip for** option specifies the number of seconds you must hold the mouse pointer over assigned cells and pins before a tooltip appears displaying the names of the assigned nodes and pins. You can type or select a number. Selecting **0** causes the tooltip to appear immediately when the pointer is over an assigned cell or pin. Selecting **Unlimited** prevents tooltips from displaying. The default value is 1.

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.

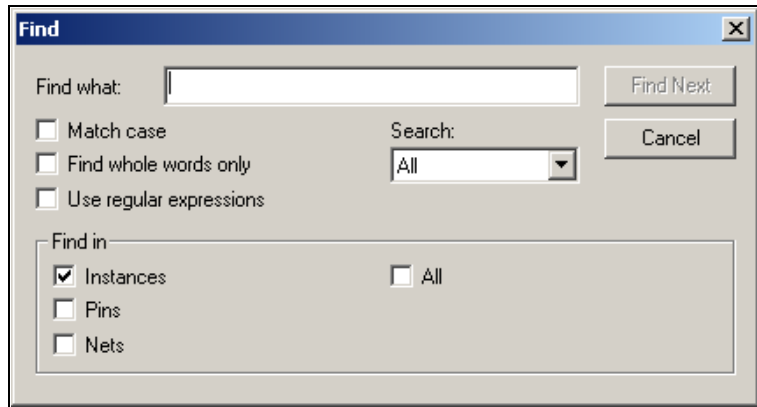
Full Screen View

Set the viewer window to fill the whole screen with **Full Screen** (View menu), by clicking **Full Screen** in the viewer toolbar, or by pressing the keyboard shortcut 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 by selecting **Find** (View menu) and clicking **Find** in the viewer toolbar, or by right-clicking in the schematic view and choosing **Find**. The **Find** dialog box, as shown in [Figure 14-14](#), is the standard search box used throughout the Quartus II software.

Figure 14–14. Find Dialog Box



For the search direction in the **Search** list, **Up** searches from the current hierarchy to the upper (parent) hierarchies, and **Down** searches from the current hierarchy to the lower (children) 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 can be seen in the viewable area (but does not zoom in to the node). To find the next 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 a JPEG File Interchange Format (.jpg) or 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). At 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 using 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 [“Page Partitioning in the Schematic View” on page 14–14](#) to control how much of your design is shown on each schematic page).

Printing

You can print your schematic page by choosing **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 [“Page Partitioning in the Schematic View” on page 14–14](#) to control how much of your design is shown on each schematic page.



Note that 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 & 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 Quartus II 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 may be able to 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 11-11
- Adaptive Logic Modules
 - Architectures with Six-Input LUTs 7-30
- Adder
 - Subtractors 7-6
 - Trees 7-28
- altera_chip_pin_lc 9-12
 - Other Devices 9-12
 - VHDL for ACEX 1K & FLEX 10KE 9-12
- altera_implement_in_eab 9-13
- altera_implement_in_esb 9-13
- altera_io_opendrain 9-13
- altera_io_powerup 9-13
- Analyze Design Problems
 - RTL 14-28
- Architecture-Specific Features 9-17, 11-14, 13-5
- Assigning a Pin 8-31
- Assignment Editor 2-18
- Assignments
 - Making 2-17
- Asynchronous
 - Clear 7-5
 - Design
 - Hazards 6-2
- Avalon Switch Fabric 3-4

B

- Back & Forward Page Viewing 14-17
- Binary Multiplexer Design 7-45
- Black Box
 - Example for Top-Level File A.v 9-38
 - in Verilog HDL 9-38
 - in VHDL 9-38, 10-26
 - Methodology 9-19, 11-16
 - Modules
 - Including Design Files 10-8
 - Top-Level File A.vhd 9-39
 - Top-Level File Verilog 10-25

- Top-Level File VHDL 10-26
- Verilog HDL 10-25, 11-27
- VHDL 11-28
- BLIS
 - LogicLock Feature 12-4
 - Using Shell Commands 12-9
- Block Root 12-3
 - Removing 12-9
- Block-Based
 - LogicLock
 - Methodology 1-4
- Block-Level Incremental Synthesis 12-2

C

- CASE
 - Assignment
 - Default or Others 7-40
 - Statement
 - Degenerate Multiplexer 7-44
 - Simple Binary-Encoded 7-39
 - Simple One-Hot-Encoded 7-39
- Chip Pin 8-27
- Clear Box Methodology 9-18, 11-15
- Clock
 - Clock & Register-Control
 - Architectural Features 6-14
 - Clock-Gating
 - Recommended Method 6-11
 - Divided 6-8
 - Frequencies 9-7
 - Gated 6-10
 - Internally-Generated 6-7
 - Multiple Clock
 - Domains 9-7
 - Multiplexed 6-9
 - Network Resources 6-14
 - Schemes 6-7
- Coding Recommendations 7-31
- Coding
 - Device-Specific Recommendations 7-25

- Combinational
 - Logic Structures 6-4
 - Loops 6-4
 - Node/Implement as Output of Logic Cell 8-10
- Command Prompt 1-8
- Compile
 - Automatically Defining Points from GUI 9-32
 - Defining Points Using Tcl or SDC 9-31
 - FPGA Design 4-6
 - Manually Defining Points from GUI 9-31
 - Set Points 9-30
 - Time 5-2
- Compiler
 - FPGA Compiler II 12-3
 - FPGA Compiler II Design Block 12-2
- Compiler Tool 2-8
- Compiling 11-5
- Connectors
 - Input 14-15
 - Output 14-16
- Constraints
 - LAB Assignments 4-13
 - Location 4-13
 - Placement 4-12
 - Timing 13-13
- Controlling Fan-Out on Data Nets 11-10
- Copying to Clipboard 14-27
- Counters
 - Ripple 6-9
- Counters 7-5
- Create Constraint Files 9-30
- Cross-Probing 9-15
 - Enabling 9-15
 - Synplify
 - Software 9-16

D

- DC FPGA
 - Environment Set Up for Altera Device
 - Families 13-3
 - Reading Design Files 13-9
- dedicated_mult 11-21
- Defaults
 - Implicit 7-41

- Degenerate Binary Multiplexer
 - Recoder Design 7-45
- Delay Chains 6-5
- Design
 - Compiling 11-6
 - Creating 12-6
 - Entry 2-14
 - Flow 9-1, 10-1, 11-1
 - Hierarchy 12-1
 - Optimization 4-10
 - Recommended Techniques 6-3
 - Reporting Information 13-16
 - Typical Flow 2-2
- Design Assistant
 - Interface 4-15
 - Settings 4-15
- Design Flow 2-13
 - Flattened versus Hierarchical Block-Based 1-1
 - One Step Process 4-6
 - RTL 14-3
 - Using DC FPGA 13-2
- Design Partitioning
 - Hierarchical 6-12
- Designs
 - Block-Based 1-2
 - Exporting 13-18
 - Targeting
 - HardCopy APEX Devices 4-14
- Device Support 2-3
- DSP
 - Block
 - Controlling Inference 10-12, 11-20
 - Blocks
 - Controlling the Inferring 9-24
 - Guidelines 10-17
 - Controlling Block Inference 11-17

E

- ECO Support 5-2
 - HDL Level 5-3
 - Netlist Level 5-5
- EDA
 - Timing Simulation 2-28
- EDIF

- Creating 10–20, 10–22, 10–27
- Creating a Project for Multiple Files 11–29
- Creating Project Including LogicLock Regions 11–25
- Generating Design with Multiple Files Using Black Boxes 11–26
- Generating Multiple Files 10–20, 10–24
- Manually Creating Multiple Files Using Black Boxes 11–26
- Embedded Software Applications 3–4
- Encoding Style 10–5

F

- Filtered Netlist
 - 14–19
 - Expanding 14–21
 - Reducing 14–21
- Find Command 14–26
- Flip-Flops 7–25
- FPGA Project
 - Close 4–6
- FSM
 - Explorer in Synplify Pro 9–10
 - Finite State Machine (FSM) Compiler 9–9
- Full Screen View 14–26
- Functional Simulation 2–20

G

- General Optimization
 - Attributes 9–10
 - Options 9–10
- Generation
 - Automatic 3–4
- Global Attribute 10–13
- Go to Net Driver 14–17

H

- HardCopy
 - APEX 20K Power Calculator 4–17
 - Design Flow 4–4
 - Design Guidelines
 - Checking Designs 4–15
 - Devices
 - Designing 4–6

- Targeting Designs 4–6
- Floorplans 4–10
- Generating Design Database 4–16
- Open Project 4–6
- Stratix Device
 - Compile 4–6
- Stratix Devices 4–3
- Stratix Power Calculator 4–18
- HDL
 - Options in Source Code 8–23
 - Read Comments 8–7
- Hierarchy
 - Considerations 12–5
 - Design Considerations 10–19, 11–23
 - Hierarchical Boundary
 - Preserve 8–15
 - List
 - 14–12
 - Selecting an Item 14–13
 - Preserving 9–11

I

- I/O
 - Assigning
 - Registers 11–9
 - Disabling I/O
 - Pad Insertion 11–9
 - Flip-Flops 8–23
 - Input/Output Delays 9–8
 - Preventing Precision RTL Synthesis from
 - Adding I/O Pad On Individual Pin 11–10
 - Preventing Precision RTL Synthesis from
 - Adding I/O Pads 11–10
 - Registers
 - Mapping 10–7
 - Setting
 - Assigning
 - Pin Numbers 11–8
- IF Statement
 - Default Conditions Explicitly Specified 7–42
 - Implicit Defaults 7–42
 - Implying Priority 7–40
- Incremental Fitting
 - Performing 1–8

- Inferring
 - Multiplier
 - DSP
 - Functions 10-11
 - Quartus II
 - Megafunctions from HDL Code 11-17
 - Memory Elements 10-10
 - RAM 10-10
 - ROM 10-11

L

- Labeling Block Roots 12-7
- Latches
 - 6-7
- Latches
 - 7-31
- Logic
 - Cells
 - Remove Redundant 8-20
 - Elements
 - Architectures with Four-Input
 - LUTs 7-29
 - Remove Duplicate 8-19
- LogicLock
 - Apply Attributes 9-33
 - Assignments 4-14
 - Block-Based Design 11-23
 - Creating Design with Separate
 - Blocks 11-24, 11-25
 - HARDCOPY_FPGA_PROTOTYPE.qsf
 - File 4-14
 - Methodology
 - Block-Based Design 9-28
 - Block-based Design 10-18
 - Migrated HardCopy Stratix .qsf File 4-14
 - Supported Constraints Example 4-14
- LogicLock Option 10-20
- LogicLock_Incremental.tcl
 - Modifications Required for Script File 10-29
- LogicLock_Interface.tcl
 - Incremental Synthesis for Script File 10-30
- LPM Functions 10-9

M

- Magnification 14-13

- Maintaining 12-6
- MAX+PLUS II
 - Converting Existing Design 2-10
 - Converting Graphic Design Files 2-11
 - Design Conversion 2-10
 - Importing Assignments 2-12
 - Look & Feel 2-6, 2-7
- Maximum Fan
 - Out 8-12, 9-10
- Megafunction
 - Inference Control 8-20
- Megafunctions 11-14, 13-5
 - Instantiating Using Port & Parameter 7-4
- Module
 - Level Attributes 10-13
 - Pool 3-7
 - Table 3-8
- Modules Constraint Table
 - Opening Block Roots 12-7
- Multiple Files 10-20, 10-27
- Multiple Files Including LogicLock
 - Regions 10-22
- Multiplexers 7-38
 - Binary 7-38
 - Buses 7-46
 - Degenerate 7-43
 - Priority 7-38, 7-39
 - Restructure 8-16
 - Restructuring Option 7-47
 - Selector 7-39
- Multiplier
 - 9-23, 11-17
 - Accumulators 10-12, 11-19
 - Adders 10-12, 11-19
 - Simple 10-11
- Multiply Accumulators
 - Multiply-Adders 8-21
 - Multiply-Adders 7-10
- Multivibrators 6-5

N

- NativeLink Integration
 - Exporting Designs 10-8, 11-12
 - Exporting Quartus II Designs 9-13
- Net Names
 - Displaying 14-26

- Netlist Files
 - Design Partitioning 1-6
 - Exporting Block-Level 12-7
 - Generating 10-8
- Netlist Optimization
 - Prevent Further 1-9
- Nets
 - Preserving 9-10
- Node Finder 2-4
- Node-Level Netlist
 - Save into Persistent Source File 1-8

O

- Optimization Strategies 10-4
- Optimization Technique 8-13
- Options
 - HDL Option 3-11
- Other Features 14-22

P

- Parallel Case 8-9
- Passing Constraints Via Scripts 10-8
- Paths
 - False 9-8
 - Multi-Cycle Paths 9-8
- Performance Estimation 4-10
- Peripherals
 - User-Defined 3-3
- Place & Route 2-22, 13-21
- Power Calculators
 - FPGAs 4-20
- Power Estimation 2-28, 4-17
- Power Tab
 - Clock 10-4
 - Global 10-4
 - Input & Output 10-5
- Power-Up
 - Don't Care 8-19
 - Level 8-18
- preserve_signal 11-21
- Printing 14-28
- Project
 - Creating 11-5
 - Creating New 2-14
 - Migrate Compiled 4-6

- Project Navigator 2-4
- Pulse Generators 6-5

Q

- Quartus II
 - Attributes 8-25, 9-11
 - Integration 10-9
 - Megafunctions 9-17, 10-9
 - Inferring from HDL Code 9-23, 13-8
 - Inferring from HDL Code 7-4
 - Instantiating 10-9
 - Instantiating and Inferring 7-1
 - Instantiating in HDL Code 7-2
 - Instantiating Using MegaWizard Plug-In Manager 7-2, 11-15
 - Instantiating Using MegaWizard Plug-In Manager 9-18
- MegaWizard
 - Generated Variation Wrapper Files
 - Black Box Methodology 13-7
 - Reading 13-7
 - Generated Verilog HDL Files
 - Black Box Megafunction Instantiation 13-7
 - Generated VHDL Files
 - Black-Box Megafunction Instantiation 13-8
 - Plug-In Manager
 - Instantiating Quartus II Megafunctions 13-7
 - Using Generated Files for Certain LPM Functions in the Synplify LPM 9-19
 - Using Generated Verilog HDL Files for Black-Box Megafunction Instantiation 9-20
 - Using Generated Verilog HDL Files for Clear Box Megafunction Instantiation 9-18
 - Using Generated VHDL Files for Black Box Megafunction Instantiation 9-21
- Probing to Source Design File 14-22
- Running the Software Manually 11-14

- Synthesis
 - 2–20
 - Attributes 8–5
 - Directives 8–4
 - Flow
 - Incremental 10–29
 - Options 8–3, 8–29
 - Results
 - Saving 13–17
- Quartus II Synthesis 12–3
- Quartus II 13–2
 - Command Reference for
 - MAX+PLUS II 2–32
 - GUI Overview 2–4
 - Logic Options 8–6
 - Simulator Tool 2–26
 - Synthesis
 - Options 8–6
- Quick Menu Reference 2–30

R

- RAM 8–22
- RAM Style 8–22
- Register Control Signals 6–15
- Register Packing 9–11
- Registers
 - Remove Duplicate 8–20
 - Secondary Control Signals 7–25
- Reports 4–15
- Reset Resources 6–14
- Resource Balancing 9–23
- Resource Sharing 10–6
- ROM 8–22
 - Inferring 9–27
- Routing
 - Preserving 1–6
- RTL Viewer 14–1

S

- Sample Verilog-1995
 - Code with a ramstyle Attribute 8–22
- Schematic
 - Exporting as JPEG or BMP 14–27
 - Following Nets Between Pages 14–15
 - Moving Between Pages 14–15

- Navigating View 14–13
- Other Features in Viewer 14–24
- Page Partitioning in View 14–14
- Selecting Item in View 14–11
- Symbols 14–5
- View 14–5
- Schematic View
 - Filtering 14–17
- Scripting Support 1–8, 8–29
- Setting
 - Constraints 11–6
 - Mapping Constraints 11–7
 - Timing Constraints 11–7
- Shift Registers 8–21
- Shift Registers 7–21
- Signal
 - Attributes for Controlling DSP Block Inference for VHDL Code 10–17
 - Attributes for Controlling DSP Block Inference in Verilog HDL Code 10–16
 - Level Attributes 10–15
- Signal Level Attribute 9–24
- Signals
 - Tri-State Signal in Verilog HDL 7–28
 - Tri-State Signal in VHDL 7–28
- Signals
 - Tri-State Signal 7–27
- Simulating
 - ModelSim 3–12
 - Other Simulators 3–12
- Simulation
 - Model & Testbench 3–6
 - Option 3–12
- Single Precision Project 11–25
- SOPC Builder
 - Ready Functions 3–3
 - Using 3–6
- Source
 - Changing Within a Block 12–8
- State Machine
 - Processing 8–14
- State Machines 7–32
- Summary 4–15
- Synchronous
 - Clock Enables 6–11
 - Design Fundamentals 6–2
 - FPGA Design Practices 6–1

Synplify

Optimization Strategies 9-6

Pro

Implementations in 9-6

Retiming 9-11

Software

Attributes for Black-Boxing 9-22

Launch 9-14

Running Quartus II from Within 9-14

Synthesis 13-13

Synthesizing the Design & Evaluating the Results 11-11

System

Contents Page 3-7

Dependency Pages 3-12

System Generation Page 3-9

System Generation 3-13

System Generation 3-6

T

Target Device

Selecting 13-11

Tcl

Console 2-4

HardCopy Migration Support 4-9

HardCopy Stratix Support 4-20

Running Script File in

LeonardoSpectrum 10-30

Script 1-8

Team-Based 1-2

Technology Map Viewer 14-2

Technology Map Viewers 14-3, 14-28

Time Stamp Synthesis 12-6

Timing

Analysis 2-23

Leonardo-Spectrum Software 10-7

Analysis Reports 11-11

Assignments 2-19

Closure Floorplan 2-25

Driven Synthesis 10-4

Models 4-10

Results

Preserving Using LogicLock Flow 1-5

Simulation 2-26

Static Analysis 4-17

Synthesis Settings 9-6

Viewing Path in Technology Map

Viewer 14-22

Tooltips 14-24

Translate Off & On 8-7

Traversing Design Hierarchy 14-16

U

Using Black Box

10-24

V

Verification 5-2

Verilog HDL

8-1, 11-18

64-Bit Long Shift Register 7-23

64-Bit Long Shift Register 7-22

Adder/Subtractor 7-6

Code with a full_case Attribute 8-9

Code with a parallel_case Attribute 8-10

Code with a useioff Attribute 8-24

Counter with Count Enable 7-5

dedicated_mult 11-21

D-Flip-Flop with Control Signals 7-26

Dual-Clock Synchronous RAM 7-15

Module Level Attributes 10-14

Multiplier Implemented in Logic 11-19

Pipelined Binary Tree 7-29

Pipelined Ternary Tree 7-30

Read Comments as HDL Example 8-8

Signal Attributes for Controlling DSP Block

Inference 9-24

Signed Multiplier 7-9

Signed Multiply-Adder 7-12

Single-Clock Synchronous RAM 7-15

State Machine Coding Example 7-34

State Machines 7-33

Synchronous ROM 7-20

Top-level Code with Black Box

Instantiation 9-20

Translate Off & On Example 8-7

Unsigned Multiplier 7-8

Unsigned Multiply Accumulator 7-11

Using MegaWizard-generated Files for Black Box Megafunction Instantiation 11-16

Using MegaWizard-generated Files for Clear

- Box Megafunction Instantiation 11–15
- Verilog HDL
 - Support 8–1
- Verilog-1995
 - Applying Chip Pin to a Bus of Pins 8–28
 - Applying Chip Pin to a Single Pin 8–28
 - Applying Quartus II Attribute to an Entity 8–27
 - Applying Quartus II Attribute to an Instance 8–26
 - HDL 8–5
- Verilog-2001 8–11
 - Applying Chip Pin to a Single Pin 8–28
 - Applying Chip Pin to Part of a Bus of Pins 8–28
 - Applying Quartus II Attribute to an Entity 8–27
 - Applying Quartus II Attribute to an Instance 8–26
 - HDL 8–6
- VHDL
 - 64-Bit Long Shift Register 7–24
 - Adder/Subtractor 7–7
 - Applying Chip Pin to a Single Pin 8–28
 - Applying Chip Pin to Part of a Bus of Pins 8–29
 - Applying Quartus II Attribute to an Entity 8–27
 - Applying Quartus II Attribute to an Instance 8–27
 - Code for syn_encoding 9–9
 - Code with a ramstyle Attribute 8–22
 - Code with a useioff Attribute 8–25
 - Counter with Synchronous Load 7–5
 - D-Flip-Flop with Control Signals 7–27
 - extract_mac 11–21
 - Inferred Dual-Port RAM 9–26
 - Inferred Dual-Port RAM Preventing Bypass Logic 9–27
 - Module Level Attributes 10–14
 - Multiplier Implemented in Logic 11–19
 - Preventing Unintentional Latch Creation 7–32
 - Read Comments as HDL Example 8–8
 - Signal Attributes for Controlling DSP Block Inference 9–25
 - Signed Multiply Accumulator 7–13

- Single-Clock Synchronous RAM with Asynchronous Read Address 7–19
- State Machine Example 7–36
- State Machines 7–36
- Synchronous ROM 7–21
- Top-level Code with Black Box Instantiation 9–21
- Translate Off & On Example 8–7
- Unsigned Multiply-Adder 7–12
- Using MegaWizard-generated Files for Black-Box Megafunction Instantiation 11–16
- Using MegaWizard-generated Files for Clear Box Megafunction Instantiation 11–16
- VQM
 - Creating a Design with Multiple Files 9–29
 - Creating a Design with Multiple Files using Multipoint Synthesis 9–30
 - Creating a Quartus II Project for Multiple Files 9–35, 9–40
 - Generating a Design with Multiple Files Using Black Boxes 9–36
 - Hierarchy & Design Considerations with Multiple Files 9–29
 - Manually Creating Multiple Files Using Black Boxes 9–36

Z

- Zooming 14–13
- Multiplier
 - 7–8
- Options
 - SDK Option 3–10
- SOPC Builder
 - Peripherals 3–2
- Verilog HDL
 - Single-Clock Synchronous RAM with Asynchronous Read Address 7–18
- VHDL
 - Counter with Synchronous Load 7–16
 - Dual-Clock Synchronous RAM 7–17
 - Signed Multiplier 7–8
 - Unsigned Multiplier 7–9



Quartus II Handbook, Volume 2

Design Implementation & Optimization



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

Copyright © 2004 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	1-2
---	-----

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
---------------------	------

Exporting and Importing Assignments	1-13
---	------

Exporting Assignments	1-14
-----------------------------	------

Importing Assignments	1-16
-----------------------------	------

Conclusion	1-18
------------------	------

Chapter 2. Command-Line Scripting

Introduction	2-1
--------------------	-----

The Benefits of Modular 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
Check Design File Syntax	2-11
Create a Project & Synthesize a Netlist Using Netlist Optimizations	2-12
Attempt to Fit a Design as Quickly as Possible	2-13
Fit a Design Using Multiple Seeds	2-13
Makefile Implementation	2-14
The QFlow Script	2-16
More Help with Quartus II Modular Executables	2-17
Conclusion	2-18

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
Nested Commands	3-3
Arithmetic	3-3
Lists	3-4
Control structures	3-4
Procedures	3-5
Quartus II Tcl API Reference	3-6
Quartus II Tcl Packages	3-6
Loading Packages	3-8
Executables Supporting Tcl	3-8
Command-Line Options (-s, -t, etc)	3-9
Run a Tcl Script	3-9
Interactive Shell Mode	3-9
Evaluate as Tcl	3-10
Using the Quartus II Tcl Console Window	3-10
Examples	3-10
Accessing Command-Line Arguments	3-11
Using the cmdline Package	3-11
PCreating Projects & Making Assignments	3-12
Compiling Designs	3-13
Extracting Report Data	3-14
Using Collection Commands	3-15
Timing Analysis	3-16
EDA Tool Assignments	3-18
Importing LogicLock Functions	3-21
Using the Quartus II Tcl Shell in Interactive Mode	3-22
Getting Help on Tcl & Quartus II Tcl APIs	3-25
Quartus II Legacy Tcl Support	3-28
References	3-28

Chapter 4. Quartus II Project Management

Introduction	4-1
Using Revisions with Your Design	4-1
Creating and Deleting Revisions	4-2
Comparing Revisions	4-3
Creating Different Versions of Your Design	4-4
Archiving Projects	4-5
Version-Compatible Databases	4-7
Scripting Support	4-8
Managing Revisions	4-8
Archiving Projects	4-9
Restoring Archived Projects	4-9
Importing and Exporting Version-Compatible Databases	4-10
Conclusion	4-10

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	5-1
I/O Assignment Planning & Analysis Design Flows	5-1
Design Flow without Design Files	5-2
Design Flow with Complete or Partial Design Files	5-4
Inputs Used for I/O Assignment Analysis	5-6
Creating I/O Assignments	5-6
Reserving Pins	5-6
Location Assignments	5-7
Assignments with the Floorplan Editor	5-8
Generating a Mapped Netlist	5-8
Running the I/O Assignment Analysis	5-9
Understanding the I/O Assignment Analysis Report	5-9
Suggested & Partial Placement	5-10
Detailed Error/Status Messages	5-11
Scripting Support	5-11
Reserving Pins	5-11
Location Assignments	5-12
Generating a Mapped Netlist	5-12
Running the I/O Assignment Analysis	5-13
Conclusion	5-13

Section III.

Area Optimization & Timing Closure

Revision History Section III-2

Chapter 6. Design Optimization for Altera Devices

Introduction	6-1
Initial Compilation	6-2
Device Setting	6-2
Timing Requirements Settings	6-2
Smart Compilation Setting	6-3
Timing Driven Compilation Settings	6-3
Fitter Effort Setting	6-4
I/O Assignments	6-5
Design Analysis	6-6
Resource Utilization	6-6
I/O Timing (including t_{PD})	6-7
f_{MAX} Timing	6-9
Compilation Time	6-11
Optimization Techniques for LUT-Based (FPGA and MAX II) Devices	6-12
Optimization Advisors	6-12
Resource Utilization Optimization Techniques (LUT-Based Devices)	6-13
Use Register Packing	6-13
Remove Fitter Constraints	6-16
Perform WYSIWYG Resynthesis for Area	6-16
Optimize Synthesis for Area	6-16
Retarget Memory Blocks	6-18
Retarget DSP Blocks	6-19
Optimize Source Code	6-19
Modify Pin Assignments or Choose a Larger Package	6-20
Use a Larger Device	6-20
Resolving Resource Utilization Issues Summary	6-20
I/O Timing Optimization Techniques (LUT-Based Devices)	6-21
Timing-Driven Compilation	6-21
Fast Input, Output, & Output Enable Registers	6-22
Programmable Delays	6-23
Using Fast Regional Clocks in Stratix Devices	6-26
Using PLLs to Shift Clock Edges	6-26
Improving Setup & Clock-to-Output Times Summary	6-27
f_{MAX} Timing Optimization Techniques (LUT-Based Devices)	6-27
Synthesis Netlist Optimizations and Physical Synthesis Optimizations	6-28
Seed	6-30
Optimize Synthesis for Speed	6-30
LogicLock Assignments	6-32
Location Assignments & Back Annotation	6-35
Optimize Source Code	6-39
Improving f_{MAX} Summary	6-40
Optimization Techniques for Macrocell-Based (MAX 7000 and MAX 3000) CPLDs	6-41
Resource Utilization Optimization Techniques (Macrocell-based CPLDs)	6-41

Use Dedicated Inputs for Global Control Signals	6-41
Reserve Device Resources	6-42
Pin Assignment Guidelines & Procedures	6-42
Resolving Resource Utilization Problems	6-45
Timing Optimization Techniques (Macrocell-based CPLDs)	6-49
Improving Setup Time	6-50
Improving Clock-to-Output Time	6-51
Improving Propagation Delay (tPD)	6-52
Improving Maximum Frequency (fMAX)	6-53
Optimizing Source Code—Pipelining for Complex Register Logic	6-53
Compilation Time Optimization Techniques	6-55
Reducing Synthesis and Synthesis Netlist Optimization Time	6-55
Reducing Placement Time	6-56
Reducing Routing Time	6-59
Scripting Support	6-59
Initial Compilation Settings	6-60
Resource Utilization Optimization Techniques (LUT-Based Devices)	6-60
I/O Timing Optimization Techniques (LUT-Based Devices)	6-61
FMAX Timing Optimization Techniques (LUT-Based Devices)	6-62
Conclusion	6-63

Chapter 7. Timing Closure Floorplan

Introduction	7-1
Design Analysis Using the Timing Closure Floorplan	7-1
Timing Closure Floorplan Views	7-1
Viewing Assignments	7-3
Viewing Critical Paths	7-5
Physical Timing Estimates	7-11
LogicLock Region Connectivity	7-12
Viewing Routing Congestion	7-15
I/O Timing Analysis Report File	7-16
f _{MAX} Timing Analysis Report File	7-19
Conclusion	7-23

Chapter 8. Netlist Optimizations and Physical Synthesis

Introduction	8-1
Synthesis Netlist Optimizations	8-2
WYSIWYG Primitive Resynthesis	8-2
Gate-Level Register Retiming	8-4
Preserving Your 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
Physical Synthesis Report	8-13
Preserving Your Physical Synthesis Results	8-14
Applying Netlist Optimization Options	8-15

Scripting Support	8-16
Synthesis Netlist Optimizations	8-16
Physical Synthesis Optimizations	8-17
Back-Annotating Assignments	8-18
Conclusion	8-18

Chapter 9. Design Space Explorer

Introduction	9-1
DSE Concepts	9-1
DSE Exploration	9-2
DSE General Information	9-2
DSE Flow	9-4
DSE Support for Altera Device Families	9-5
DSE Exploration	9-6
DSE Project Settings	9-6
DSE Project Settings	9-6
Search for Best Area or Performance Options	9-7
Advanced Search Option	9-7
Performing an Advanced Search in Design Space Explorer	9-7
Allow LogicLock Region Restructuring	9-8
Exploration Space	9-8
Optimization Goal	9-12
Search Method	9-13
DSE Flow Options	9-13
Continue Exploration Even if Base Compile Fails	9-13
Run Quartus Assembler During Exploration	9-13
Archive All Compiles	9-14
Save Exploration Space to File	9-14
Stop Flow After Time	9-14
Stop Flow After Gain	9-14
DSE Advanced Information	9-15
Computer Load Sharing in DSE Using Distributed Exploration Searches	9-15
Creating Custom Spaces for DSE	9-16
Conclusion	9-21

Chapter 10. LogicLock Design Methodology

Introduction	10-1
Improving Design Performance	10-1
Preserving Module Performance	10-2
Designing with the LogicLock Feature	10-2
Creating LogicLock Regions	10-2
Floorplan Editor View	10-9
LogicLock Region Properties	10-10
Hierarchical (Parent and/or Child) LogicLock Regions	10-11
Assigning LogicLock Region Content	10-13
Tcl Scripts	10-15
Quartus II Block-Based Design Flow	10-16

Additional Quartus II LogicLock Design Features	10–22
LogicLock Restrictions	10–30
Constraint Priority	10–30
Placing LogicLock Regions	10–30
Placing Memory, Pins & Other Device Features into LogicLock Regions	10–32
Back-Annotating Routing Information	10–33
Exporting Back-Annotated Routing in LogicLock Regions	10–33
Importing Back-Annotated Routing in LogicLock Regions	10–35
Scripting Support	10–36
Initializing and Uninitializing a LogicLock Region	10–37
Creating or Modifying LogicLock Regions	10–37
Obtaining LogicLock Region Properties	10–37
Assigning LogicLock Region Content	10–37
Prevent Further Netlist Optimization	10–38
Save a Node-level Netlist into a Persistent Source File (.vqm)	10–38
Exporting LogicLock Regions	10–39
Importing LogicLock Regions	10–39
Setting LogicLock Assignment Priority	10–39
Assigning Virtual Pins	10–40
Back-Annotating LogicLock Regions	10–40
Conclusion	10–40

Chapter 11. Timing Closure in HardCopy Devices

Introduction	11–1
Timing Closure	11–1
Placement Constraints	11–3
Location Constraints	11–3
Location Array Block (LAB) Assignments	11–3
LogicLock Assignments	11–4
Tutorial	11–5
Minimizing Clock Skew	11–5
Checking the HardCopy Device Timing	11–7
Clock Definitions	11–7
Primary Input Pin Timing	11–8
Primary Output Pin Timing	11–9
Combinatorial Timing	11–10
Timing Exceptions	11–11
Correcting Timing Violations	11–11
Hold-Time Violations	11–11
Setup-Time Violations	11–16
Timing ECOs	11–21
Conclusion	11–22

Chapter 12. Synplicity Amplify Physical Synthesis Support

Software Requirements	12–1
Amplify Physical Synthesis Concepts	12–1
Amplify-to-Quartus II Flow	12–2

Initial Pass: No Physical Constraints	12-3
Iterative Passes: Optimizing the Critical Paths	12-5
Using the Amplify Physical Optimizer Floorplans	12-5
Multiplexers	12-7
Independent Paths	12-8
Feedback Paths	12-9
Starting and Ending Points	12-9
Utilization	12-11
Detailed Floorplans	12-11
Forward Annotating Amplify Physical Optimizer Constraints into the Quartus II Software	12-12
Altera Megafunctions Using the MegaWizard Plug-In Manager with the Amplify Software	12-13
Conclusion	12-14

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: *June 2004*
Part number: *qii52001-2.0*
- Chapter 2. Command-Line Scripting
Revised: *June 2004*
Part number: *qii52002-2.0*
- Chapter 3. Tcl Scripting
Revised: *August 2004*
Part number: *qii52003-2.0*
- Chapter 4. Quartus II Project Management
Revised: *June 2004*
Part number: *qii52012-1.0*
- Chapter 5. I/O Assignment Planning & Analysis
Revised: *June 2004*
Part number: *qii52004-2.0*
- Chapter 6. Design Optimization for Altera Devices
Revised: *June 2004*
Part number: *qii52005-2.0*
- Chapter 7. Timing Closure Floorplan
Revised: *June 2004*
Part number: *qii52006-2.0*
- Chapter 8. Netlist Optimizations and Physical Synthesis
Revised: *June 2004*
Part number: *qii52007-2.0*
- Chapter 9. Design Space Explorer
Revised: *June 2004*
Part number: *qii52008-2.0*

Chapter 10. LogicLock Design Methodology

Revised: *August 2004*Part number: *qii52009-2.1*

Chapter 11. Timing Closure in HardCopy Devices

Revised: *June 2004*Part number: *qii52010-2.0*

Chapter 12. Synplicity Amplify Physical Synthesis Support

Revised: *February 2004*Part number: *qii52011-1.0*



About this Handbook

This handbook provides comprehensive information about the Altera® Quartus®II design software, version 4.1.

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	lit_req@altera.com (1)	lit_req@altera.com (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} , lqdesigns 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>.pdf 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.
	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.

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 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 [Chapters 1 to 4](#).

Chapter(s)	Date / Version	Changes Made
1	June 2004 v2.0	<ul style="list-style-type: none">• Updates to tables, figures.• New functionality in the Quartus software version 4.1.
	Feb. 2004 v1.0	Initial release.
2	June 2004 v2.0	<ul style="list-style-type: none">• Updates to tables, figures.• New functionality in the Quartus software version 4.1.
	Feb. 2004 v1.0	Initial release.
3	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 software version 4.1.
	Feb. 2004 v1.0	Initial release.
4	June 2004 v1.0	Initial release.

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 their performance requirements. This complexity is compounded by the increasing density and associated pin counts of current FPGAs. To successfully implement a complex design in the latest generation of FPGAs, designers must also make a large number of pin assignments that include the pin locations and I/O standards.

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
- 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, you can 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 launch the Assignment Editor from the 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 itself 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. If you make an assignment in the Quartus II software, such as in the Tcl console or in the Floorplan Editor, the Assignment Editor reloads the new assignments from memory. If you modify the Quartus II Settings File (.qsf) outside the Quartus II software and you select the Assignment Editor window, the Assignment Editor reloads the QSF.



If the QSF is being edited while the project is open, Altera recommends that you perform a Save Project (File menu) to ensure that you are editing the latest QSF file.

In either case, the **Messages** window displays the following message:

```
Info: Assignments reloaded -- assignments updated
      outside Assignment Editor
```

The assignments you make in the Assignment Editor, Floorplan Editor, or with Tcl are stored in memory. Use 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

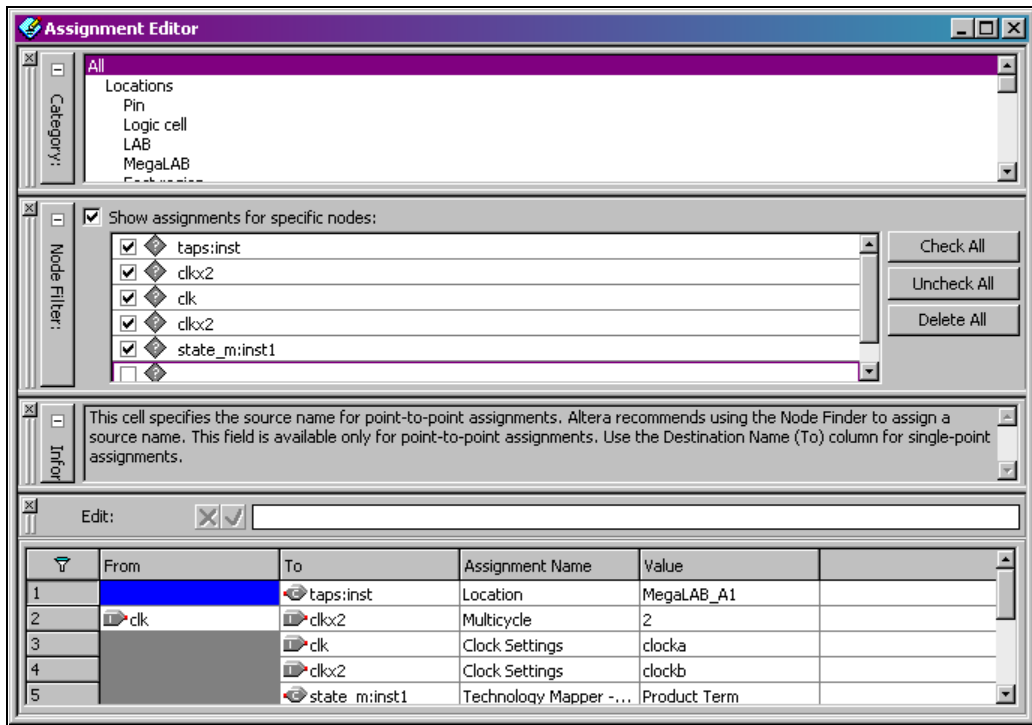


Table 1–1. Assignment Editor Bar Descriptions

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 **tsu** in the category list. If you select **All** in the **Category** bar, the Assignment Editor displays all assignments. See Figures 1–2 and 1–3 below.

Figure 1–2. All Selected in the Category List

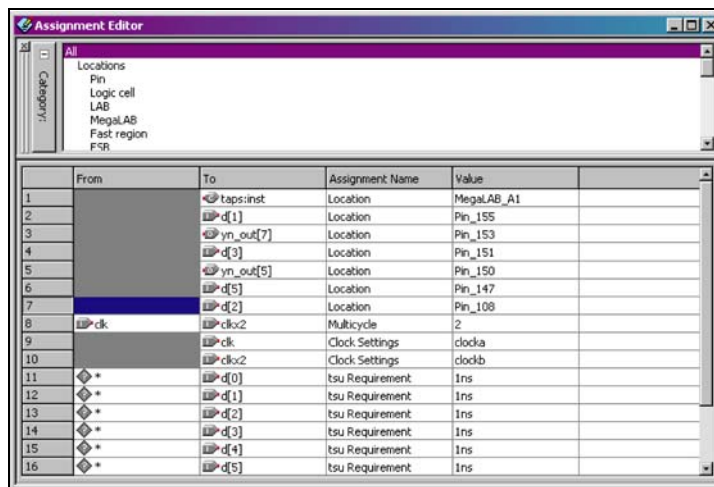
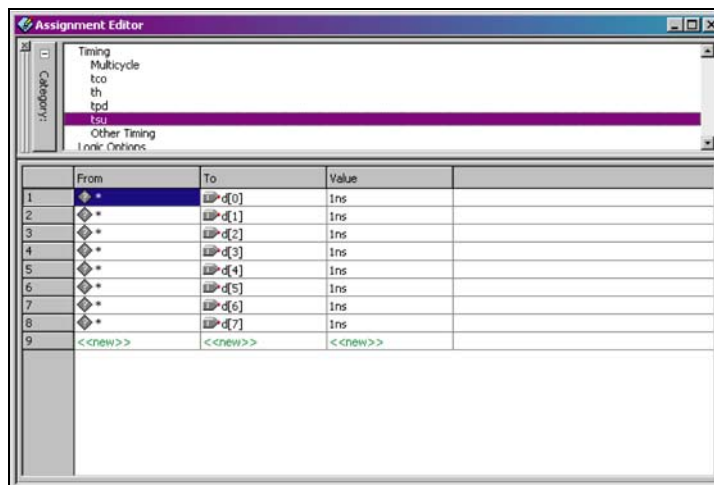


Figure 1–3. t_{SU} Selected in Category List



When you collapse the **Category** bar, four shortcut buttons appear so you can select between various preset categories (see [Figure 1-4](#)).

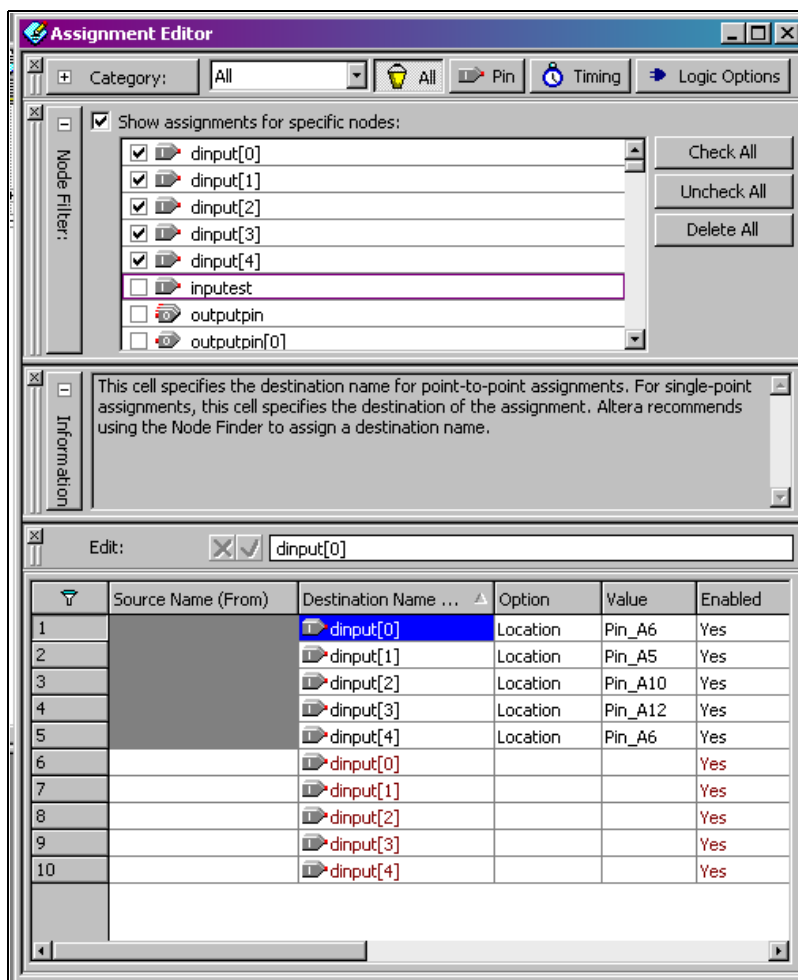
Figure 1-4. Category Bar



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.

In [Figure 1-5](#), when selecting all the bits of the `dinput` bus, all unrelated assignments are filtered out.

Figure 1–5. Using the Node Filter 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 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 (see [Figure 1-7](#)) and click the checkmark icon (Accept).

Figure 1-6. Edit Bar Selection

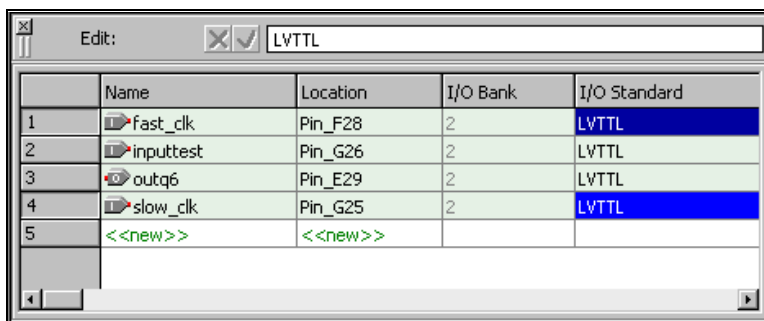
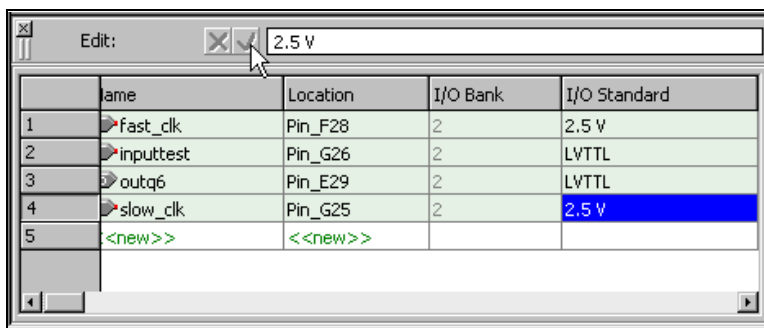


Figure 1-7. Edit Bar Change



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 to cause your node name to appear in the Assignment Editor.

You can also open other windows from the Assignment Editor. From a node listed in the Assignment Editor spreadsheet, you can go to the location of the node in any of the following windows: Timing Closure Floorplan, Last Compilation 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 the “[Dynamic Syntax Checking](#)” on page 1-9 for more information.

There are many ways to select or enter nodes into the spreadsheet, including: the Node Finder, the **Node Filter** bar, the **Edit** bar, or by directly typing the node name into the cell in the spreadsheet. A node type icon 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:

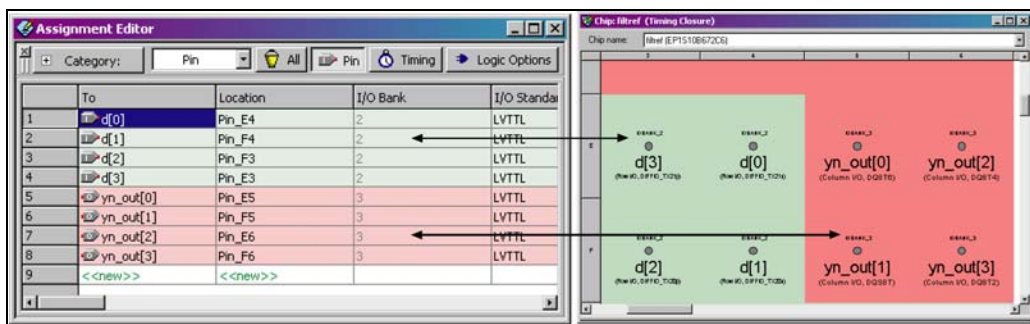
- All timing assignments
- Point-to-point global signal assignments (applicable to Stratix and Stratix II families)

- 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](#)), allowing 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



Auto-fill pin names are supported in the spreadsheet if you have performed analysis and synthesis. Auto-fill pin locations are also supported in the spreadsheet if **Pin** is selected in the **Category** bar.

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 on page 1–10](#)). You can customize the colors in the **Options** dialog box (Tools menu).



For more information, see the Quartus II Help.

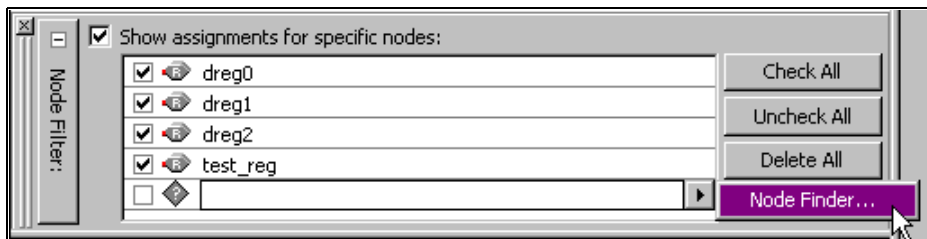
Table 1–2. Description of the Text Color in the Spreadsheet

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

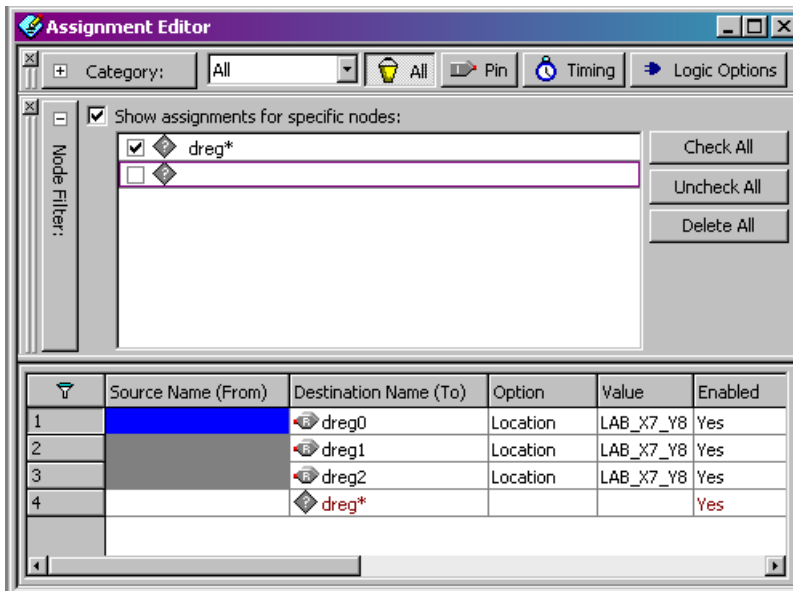
The **Node Filter** bar provides flexibility in how you view and make your settings. The **Node Filter** bar contains a list of node filters. To create a new entry, use the Node Finder or manually type the node name. Double-click an empty row in the **Node Filter** list and then click on the arrow to 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 dreg* into the Node Filter list to view all assignments for dreg0, dreg1, and dreg2 (see [Figure 1–11](#)).

Figure 1–11. Using the Node Filter Bar with Wildcards

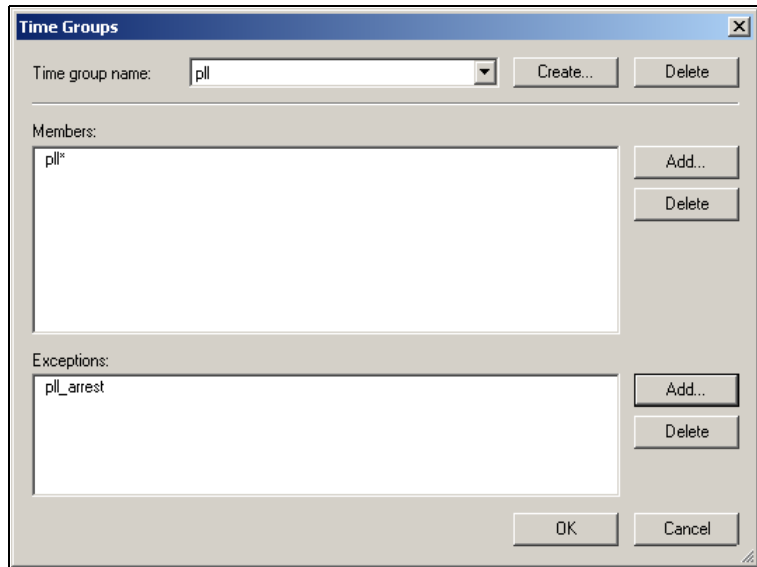


Using Time Groups

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 complex timing assignments to a large number of nodes.

To create a time group, open the **Time Groups** dialog box by selecting **Time Groups** (Assignments menu). You can add and exclude members of each time group with wild cards in the Node Finder (See [Figure 1–12 on page 1–12](#)).

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. 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.

For example, the **Comments** and **Enabled** columns are hidden when the Quartus II software is first started.

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

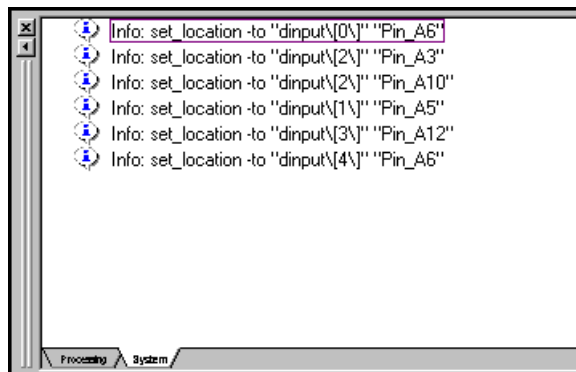
Whether you use the Assignment Editor or another feature to create your design's assignments, you can export them all to a Tcl file. You can then use the Tcl file to re-apply all the settings or to archive your assignments. Choose **Export** (File menu) to export your assignments to a Tcl script.



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 use 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*.

Exporting and Importing Assignments

With the **Export Assignments** and **Import Assignments** dialog boxes, you can export your Quartus II assignments to a Quartus II Settings file (.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 .csv or 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.

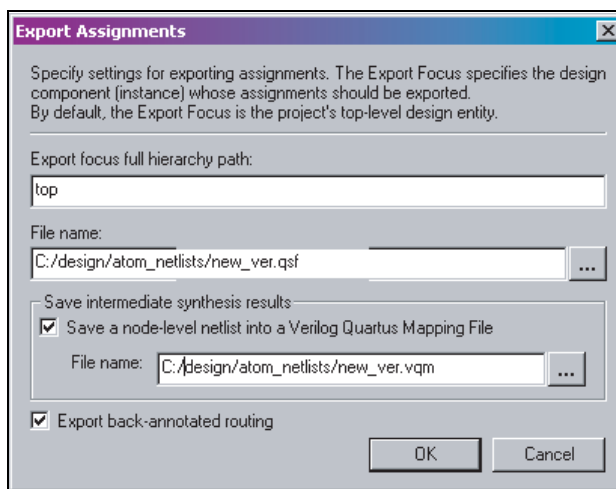
Exporting Assignments

The **Export Assignments** dialog box is used 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 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

	To	Location	I/O Bank	I/O Standard	General Function	Special Function	Reserved	SignalProbe Source
1	clk	PIN_K5	1	LVTTL	Dedicated Clock	CLK0/LVDSCLK1p		
2	button	PIN_W3	4	LVTTL	Column I/O	LVDS128p		
3	q[0]	PIN_E14	2	LVTTL	Column I/O	LVDS56n		
4	q[1]	PIN_E13	2	LVTTL	Column I/O	LVDS56p		
5	q[2]	PIN_C14	2	LVTTL	Column I/O	LVDS55n/DQ0T4		
6	q[3]	PIN_D14	2	LVTTL	Column I/O	LVDS55p/DQ0T5		
7	q[4]	PIN_E12	2	LVTTL	Column I/O	LVDS52n		
8	q[5]	PIN_F12	2	LVTTL	Column I/O	LVDS52p		
9	q[6]	PIN_B3	2	LVTTL	Column I/O	LVDS32n/DEV_OE		
10	q[7]	PIN_B14	2	LVTTL	Column I/O	LVDS57p		

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,LVDS128p,,
q[0],PIN_E14,2,LVTTL,Column I/O,LVDS56n,,
q[1],PIN_E13,2,LVTTL,Column I/O,LVDS56p,,
q[2],PIN_C14,2,LVTTL,Column I/O,LVDS55n/DQ0T4,,
q[3],PIN_D14,2,LVTTL,Column I/O,LVDS55p/DQ0T5,,
q[4],PIN_E12,2,LVTTL,Column I/O,LVDS52n,,
q[5],PIN_F12,2,LVTTL,Column I/O,LVDS52p,,
q[6],PIN_B3,2,LVTTL,Column I/O,LVDS32n/DEV_OE,,
q[7],PIN_B14,2,LVTTL,Column I/O,LVDS57p,,
```

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 on the “...” box (Browse) to navigate to the assignment file.
3. When the **Select File** dialog box opens, select the file, and click **Open** to close the **Select File** dialog box.
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 same format as it was when exported.

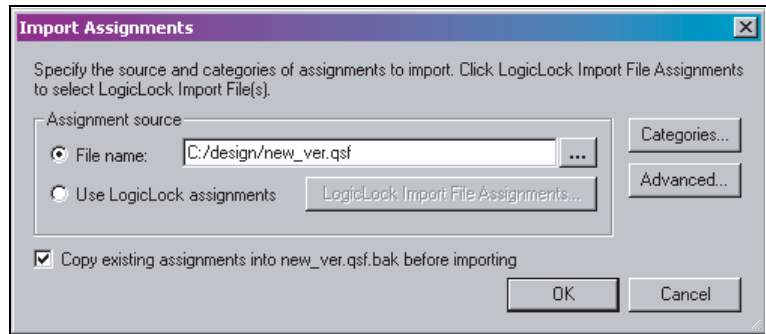
When using the Logiclock flow methodology to import assignments, follow these steps:

1. Choose **Import Assignments** (Assignments menu).
2. Select the **Use LogicLock assignments** button, and click on the **LL_IMPORT_FILE Assignments...** box.
3. When the **LogicLock Import File Assignments** window opens, select the LogicLock import file assignments to use for importing, and click **OK** to close the window.



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 copy of your assignments before importing new assignments by selecting the checkbox for **Copy existing assignments into <revision name>.qsf.bak before importing** option.

Figure 1–16. Import Assignments Dialog Box

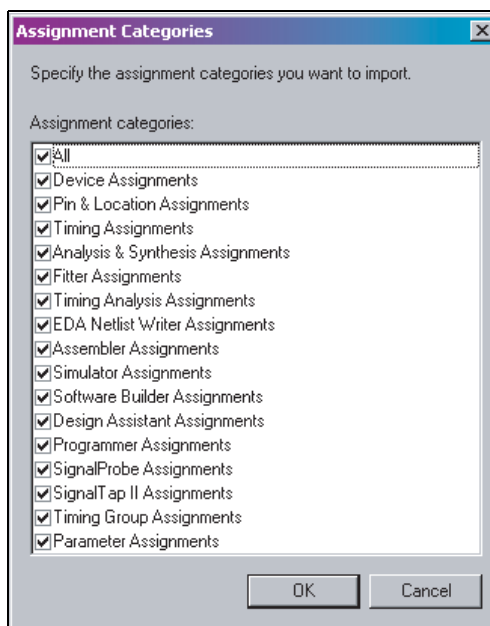
When importing assignments from a file, you can choose which assignment categories to import by following these steps:

1. Click **Categories** in the **Import Assignments** dialog box.
2. Select 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, refer to the Quartus II software 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.



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*.



2. Command-Line Scripting

qii52002-2.0

Introduction

FPGA design software that is easy to integrate into a design flow saves time and improves productivity. The Altera® Quartus® II software provides designers with modular executables for each step in an FPGA design flow to make the design process customizable and flexible.

The benefits provided by modular 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 modular executables are also completely compatible with the Quartus II graphical user interface (GUI), allowing you to use the exact combination of tools you prefer.

This chapter describes how to take advantage of Quartus II modular executables, and provides several examples of their use in certain design situations.

The Benefits of Modular Executables

The Quartus II modular 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 terms of 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 mature workstations with low amounts of memory.

Modular executables also provide command-line control over each step of the design flow. Each modular executable has options to control commonly-used software settings. Each modular executable also provides detailed, built-in help describing its function, available options, and settings.

Modular executables allow for easy integration with scripted design flows. It is simple to create scripts in any language with a series of modular executable commands. These scripts can be batch-processed, allowing for integration with distributed computing in server farms. The Quartus II modular 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.

Modular 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 modular 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 modular 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 modular 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>/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` command.

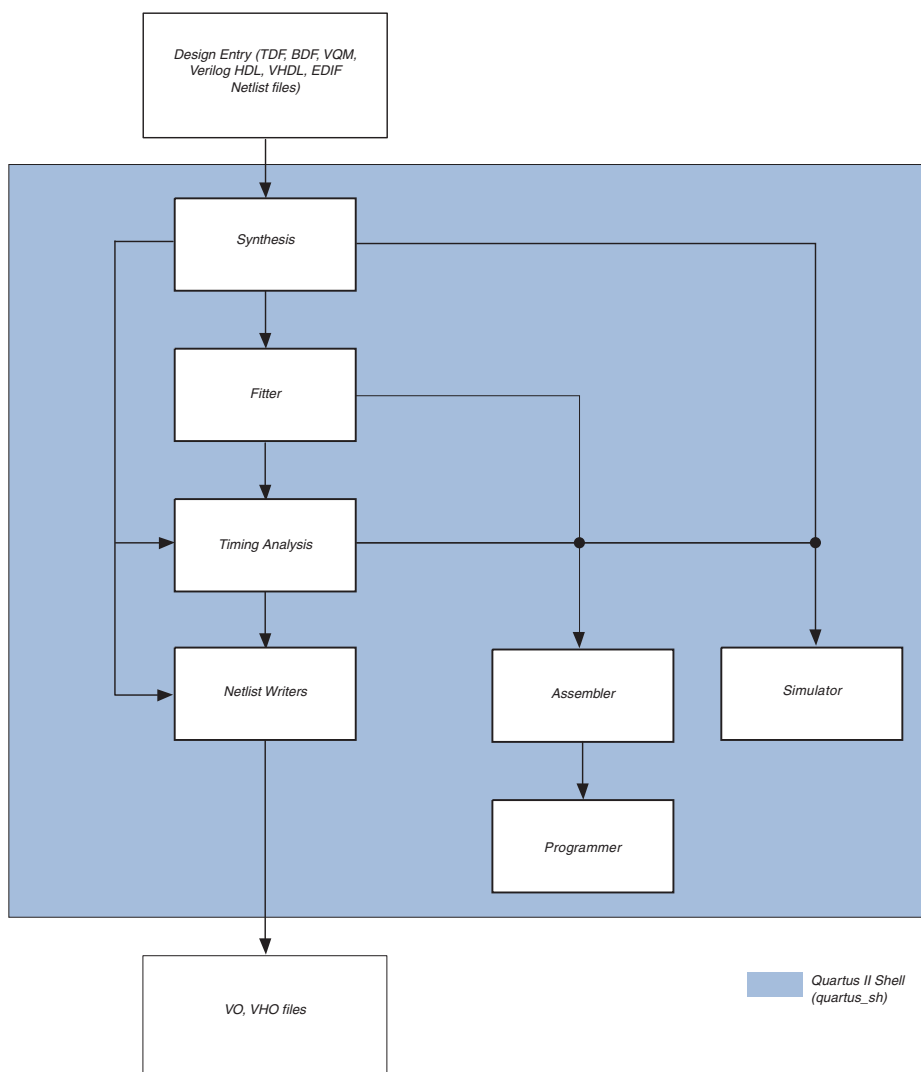
The `quartus_asm filtref` command creates programming files for the **filtref** project.

These four commands can be 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



Modular executables are provided for each stage in the design flow shown in [Figure 2–1](#). Additional modular executables are provided for specific tasks. [Table 2–1](#) lists each Quartus II modular executable and provides a brief description of its function.

Table 2–1. Quartus II Modular 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	<p>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.</p> <p>Quartus II Analysis & Synthesis must be run successfully before running the Fitter.</p>
Timing Analyzer quartus_tan	<p>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 Tcl support.</p> <p>Quartus II Analysis & Synthesis or the Fitter must be run successfully before running the Timing Analyzer.</p>
Assembler quartus_asm	<p>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 (.tff), and Raw Binary Files(.rbf), from a successful fit (that is, place-and-route).</p> <p>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.</p> <p>The Quartus II Fitter must be run successfully before running the Assembler.</p>

Table 2–1. Quartus II Modular Executables & Descriptions (Part 2 of 3)

Executable	Description
Design Assistant quartus_drc	<p>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.</p> <p>The Design Assistant supports designs that target any Altera device supported by the Quartus II software, except MAX® 3000 and MAX 7000 devices.</p> <p>Quartus II Analysis & Synthesis or the Fitter must be run successfully before running the Design Assistant.</p>
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 valid 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>

Table 2–1. Quartus II Modular Executables & Descriptions (Part 3 of 3)

Executable	Description
Convert Programming File quartus_cpf	The Quartus II Convert Programming File module converts one programming file format to a different possible format. Make sure you specify valid options and an input programming file to generate the new requested programming file format.
Quartus Shell quartus_sh	The Quartus II Shell acts as a simple Quartus II Tcl interpreter. The Shell has a smaller memory footprint than the other command-line executables that support Tcl: <code>quartus_tan</code> , <code>quartus_cdb</code> , and <code>quartus_sim</code> . 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.

Text-Based Report Files

Each modular 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 modular executable.

Report file names contain the revision names and name of the modular executable that generated the report file. For example, the report file name format is *<revision name>.<modular executable>.rpt*. For example, using the **quartus_fit** modular executable to place-and-route a project with the revision name **design_top** generates a report file named **design_top.fit.rpt**. Likewise, using the **quartus_tan** modular 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 Modular Executables” on page 2–17](#).

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` command 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 modular executable. For more information about the modular executable options, use the help information integrated with the modular executables. Access help for a modular executable using the `-h` option. For example, to view help for the `quartus_map` modular 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_retiming[=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 modular executables is also available by typing `quartus_sh --qhelp` at a command prompt. For more information, see [“More Help with Quartus II Modular Executables”](#) on page 2-17.

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 modular executables that are run after this update will 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 modular 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 will be 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 the

--import_settings_files option on causes a modular executable to read assignments from the Quartus II settings file instead of from the compiler database files. Turning the --export_settings_files option on causes a modular 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)	<ol style="list-style-type: none"> 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	<ol style="list-style-type: none"> 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` command).

```
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 modular 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 modular 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.

Check Design File Syntax

This shell script example assumes that the Quartus II software tutorial project called **fir_filter** exists in the current directory. (This project exists in the *<Quartus II directory>/qdesigns/fir_filter* directory unless the Quartus II software tutorial files are not installed.) The `--analyze_file` option specifies each file on which to perform a syntax check. The script checks the exit code of the `quartus_map` executable to determine whether there was an error during the syntax check. Files with syntax errors are added to the `FILES_WITH_ERRORS` variable, and when all files have been checked for syntax, the script prints a message indicating whether there were any syntax errors. Any options that are not specified use the values from the project database. If not specified there, then 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.

This 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_retiming=on` and `--enable_wysiwyg_resynthesis=on` options allow the technology mapper to optimize the design using gate-level register retiming and technology remapping.



For more details about register retiming, technology remapping, and other netlist optimization options, consult the Quartus II Help.

The `--part` option tells the technology mapper to target an EP20K600EBC652-1X device. To create the project and synthesize it using the netlist optimizations described above, type the following command at a command prompt:

```
quartus_map top --source=top.edf --enable_register_retiming=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 modular 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. The following example is one way of implementing a makefile with modular executables.

```

#####
# 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 which 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

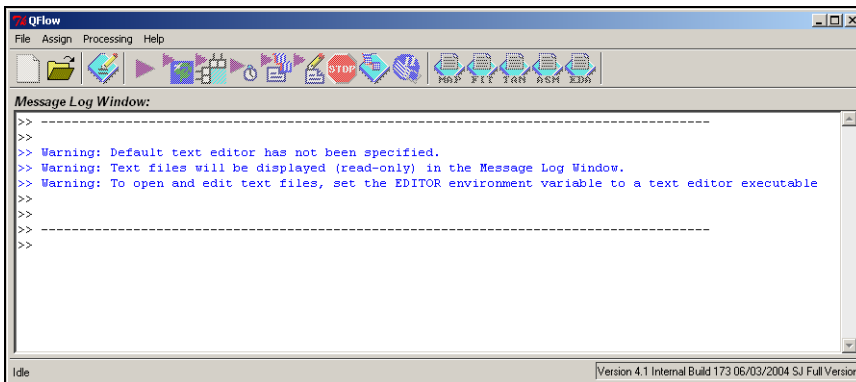
A Tcl/Tk-based graphical interface called QFlow is included with the modular executables. Designers can use the QFlow interface to open projects, launch some of the modular executables, view report files, and make some global project assignments. The QFlow interface can run the following modular executables:

- **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



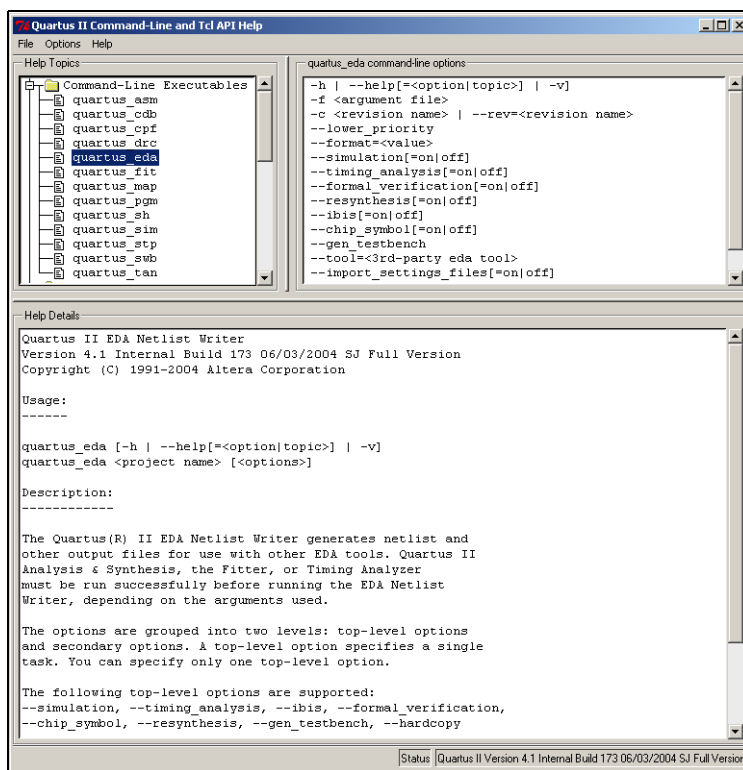


The QFlow script is located in the *<Quartus II directory>/bin/tcl_scripts/qflow/* directory.

More Help with Quartus II Modular Executables

More information on modular 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 modular 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.

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 electronic design automation (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 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 3–28](#).

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 version 4.1 supports Tcl/Tk version 8.4, supplied by the Tcl DeveloperXchange at <http://tcl.activestate.com>.

Tcl Scripting Basics

This section is a brief introduction to Tcl, an interpreted scripting language. The core 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. There are many more Tcl commands and features not covered in this brief introduction.



For more information about Tcl scripting, consult any of the “References” on page 3–28.

Hello World Example

This example shows the basic “Hello world” in Tcl.

```
puts "Hello world"
```

Use double quotation marks to group the words `hello` and `world` as one argument. Double quotation marks allow substitutions to occur in the group. Substitutions can be simple variable substitutions, or the result of running a nested command, described later in this section. 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. This 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. This example also prints "Hello world".

```
set a Hello
set b world
puts "$a $b"
```

Nested Commands

Use square brackets to evaluate nested commands. 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]
```

There are many other operations the string command can perform. Refer to the references at the end of this chapter for more information.

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 `a` to the sum of 1 and the square root of 2.

```
set a [expr { 1 + sqrt(2) }]
```

Tcl also supports boolean operators such as `&` (AND), `|` (OR), `!` (NOT), and comparison operators such as `<` (less than), `>` (greater than), and `==` (equal to).



For a complete list of supported operators, refer to [“References” on page 3–28](#).

Lists

A Tcl list is a series of values. Supported list operations include creating lists, appending lists, extracting 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 }
```

This example prints the 0th element of the list stored in `a`.

```
puts [lindex $a 0]
```

This example sets `b` to the length of the list stored in `a`.

```
set b [llength $a]
```

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
```

Quartus II Tcl API Reference

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.

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 (Part 1 of 2)	
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

Table 3–1. Tcl Commands Grouped in Packages, by Function (Part 2 of 2)

Package Name	Package Description
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 are 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, 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. Refer to 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. You should use the `quartus_sh` executable if you run Tcl scripts with only project management and assignment commands, or need a Quartus II command-line executable with a small memory footprint.



For more information about these command-line executables, refer to the *Command-Line Scripting* chapter in Volume 2 of the *Quartus II Handbook*.

Command-Line Options (-s, -t, etc)

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
<code>-t <script file></code> <code>[<script args>]</code>	Run the specified Tcl script with optional arguments
<code>-s</code>	Open the executable in the interactive Tcl shell mode
<code>--tcl_eval</code> <code><tcl command></code>	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 version 4.0 and earlier does not support the `argv` variable. In those versions of the software, script arguments are 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 Quartus II graphical user interface (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, refer to 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.

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:

```
quartus_sh -t print_args.tcl my_project 100MHz↵

    The value at index 0 is my_project

    The value at index 1 is 100MHz
```

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 code 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:

```
quartus_sh -t print_cmd_args.tcl -project my_project -frequency 100MHz↵

    The project name is my_project
    The frequency is 100MHz
```



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*

PCreating 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 generate a Tcl file with all of your assignments automatically. 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 script is a compilation script for the finite impulse response (FIR) filter example project used in the Quartus II Tutorial. It shows how to set global, location, and instance assignments for a project followed by a complete project compilation using the `::quartus::flow` package.

```
# This Tcl file works with quartus_sh.exe
# This Tcl file will compile the Quartus II tutorial fir_filter
# design
# set the project_name to fir_filter
# set revision to filtref
set project_name fir_filter
set revision_name filtref

# Create a new project and open it
# Project_name is project name
# No need to explicitly require the ::quartus::project package,
# because it's automatically loaded by quartus_sh
if {![project_exists $project_name]} {
    project_new -revision $revision_name $project_name;
} else {
    project_open -revision $revision_name $project_name;
}

#----- Make global assignments -----#

# add design files to project
# When the revision name is the same as the project name
# adding design files can be skipped
#set_global_assignment -name "BDF_FILE" "filtref.bdf"
#set_global_assignment -name "VERILOG_FILE" "acc.v"
#set_global_assignment -name "VERILOG_FILE" "accum.v"
#set_global_assignment -name "VERILOG_FILE" "hvalues.v"
#set_global_assignment -name "VERILOG_FILE" "mult.v"
```

```
#set_global_assignment -name "VERILOG_FILE" "state_m.v"
#set_global_assignment -name "VERILOG_FILE" "taps.v"

set_global_assignment -name FAMILY Cyclone

#----- project compilation -----#

# The project is compiled here to see ESB placement following
# what is done in the tutorial
load_package flow
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.



You can also use the Tcl `exec` command to perform command-line system calls. However, Altera recommends using the `qexec` command to avoid limitations with Tcl version 8.3. Whether using `exec` or `qexec`, use caution when making 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.

Report panels are arranged hierarchically, 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 can 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 collection, `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, refer to 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 the timing analysis discovers a negative slack for one of the clocks. The second part of the script adds a multiclock assignment from `clk` to `clkx2` and re-analyzes the design as a multi-clock, multiclock 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 by using the `create_timing_netlist` and `report_timing` Tcl commands.



You must compile the project before running the script example below.

```
# 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.

The global options used for interface to EDA tools in the Quartus II software are shown below:

- `EDA_DESIGN_ENTRY_SYNTHESIS_TOOL`
- `EDA_SIMULATION_TOOL`
- `EDA_TIMING_ANALYSIS_TOOL`
- `EDA_BOARD_DESIGN_TOOL`
- `EDA_FORMAL_VERIFICATION_TOOL`
- `EDA_RESYNTHESIS_TOOL`

By default, these project options are set to <none>. Table 3–5 lists the EDA interface options available in the Quartus II software. Enclose interface assignment options that contain spaces in quotation marks.

Table 3–5. EDA Interface Options	
Option	Acceptable Values
Design Entry (EDA_DESIGN_ENTRY_SYNTHESIS_TOOL)	Design Architect Design Compiler FPGA Compiler FPGA Compiler II FPGA Compiler II Altera Edition FPGA Express LeonardoSpectrum LeonardoSpectrum-Altera (Level 1) Synplify Synplify Pro ViewDraw Precision Synthesis Custom
Simulation (EDA_SIMULATION_TOOL)	ModelSim (VHDL output from the Quartus II software) ModelSim (Verilog HDL output from the Quartus II software) ModelSim-Altera (VHDL output from the Quartus II software) ModelSim-Altera (Verilog HDL output from the Quartus II software) SpeedWave VCS Verilog-XL VSS NC-Verilog (Verilog HDL output from the Quartus II software) NC-VHDL (VHDL output from the Quartus II software) Scirocco (VHDL output from the Quartus II software) Custom Verilog HDL Custom VHDL
Timing Analysis (EDA_TIMING_ANALYSIS_TOOL)	Prime Time (VHDL output from the Quartus II software) Prime Time (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 VHO file 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
```

There are custom options 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.

Importing LogicLock Functions

The following Tcl script shows how a LogicLock function can be imported into a project. This example is based on the LogicLock tutorial design **topmult**. The script assumes that the Verilog Quartus Mapping file (**.vqm**) named **pipemult.vqm** and the Quartus II Setting File named **pipemult.qsf** have been generated already and placed in the **topmult** project directory. To import LogicLock regions into a project, the **quartus_cdb** executable must be used.

```
# Tcl file created for quartus_cdb to import LogicLock
# pipemult.vqm and pipemult.qsf into the topmult project
# This Tcl script assumes that pipemult.vqm and pipemult.qsf
# have been generated in the lockmult project.

# Since ::quartus::flow is not pre-loaded
# by quartus_cdb, load this package now
# before using the flow Tcl API
# Type "help -pkg flow" to view information
# about the package
load_package flow

set required_fmax 150.00MHz

set project_name topmult

# $project_name contains the project
# name, in this case fir_filter
# Require package ::quartus::project
load_package project

project_open $project_name

#----- Make global assignments -----#

# remove bdf file from project
set_global_assignment -name "BDF_FILE" "pipemult.bdf" -remove
# add VQM file to project
set_global_assignment -name "VQM_FILE" "pipemult.vqm"

# analyze design with VQM file
execute_module -tool map
```

```
# import LogicLock constraints
load_package logiclock
initialize_logiclock

# imports the pipemult.qsf file to the project topmult.qsf
logiclock_import -no_pins

uninitialize_logiclock

# compile entire design
execute_flow -compile

#----- Report Fmax from report -----#
load_package report
load_report
set actual_fmax [get_timing_analysis_summary_results -\
clock_setup clk -actual]
puts ""
puts "-----"
puts "Required Fmax: $required_fmax Actual Fmax: $actual_fmax"
puts "-----"

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 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 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 picked up from the hierarchy tree automatically.

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 can be used 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  
  
-----
```

```
tcl>
```

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 happens when 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 3–6 summarizes the help options available in the Tcl environment.

Table 3–6. 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></code> <code>[-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> ↵.</p> <p>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>
<code><command_name> -h</code> or <code><command_name> -help</code>	<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>
<code>package require</code> <code>::quartus::<<package name></code> <code>[<version>]</code>	<p>To load a Quartus II Tcl package with the specified version. If <code><version></code> 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>

Table 3–6. Help Options Available in the Quartus II Tcl Environment (Part 2 of 2)

Help Command	Description
<pre>help -cmd <command name> [-version <version number>] or <command name> -long_help</pre>	<p>To view long help for a Quartus II Tcl command. Only <i><command name></i> -long_help"requires that the associated Tcl package is loaded.</p> <p>If you do not specify the -version 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” on page 3–28 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 system 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

The Quartus II software version 3.0 and later 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
- Tcl Developer Xchange at <http://tcl.activestate.com>

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 Version-Compatible Database feature allows databases to be compatible across different versions of the Quartus II software, thus avoiding unnecessary recompilations.

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. Working on a revision based on another revision allows you to revert to the original revision if you are not satisfied with the results from the new 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, and perform further optimizations until the design meets all design requirements.

Creating and Deleting Revisions

All Quartus II assignments and settings are stored in the Quartus Settings File (QSF). Each time you create a new revision 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). Revisions are managed with the **Revisions** dialog box, allowing you to set the current revision, create, delete, and compare revisions in a project.

To create a revision:

1. If you have not already done so, create a new project or open an existing project.
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**.
5. In the **Create Revision** dialog box, 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 revision in Step 3, then select the revision in the **Based on revision** list.

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. If you want, 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**.
10. Click **OK**.
11. In the **Revisions** dialog box, click **Close**.

To delete a revision that is not a design's current revision:

1. If you have not already done so, open an existing project.
2. Choose **Revisions** (Project menu).
3. In the **Revisions** list, select the revision you want to delete.
4. Click **Delete**.
5. Click **Close**.

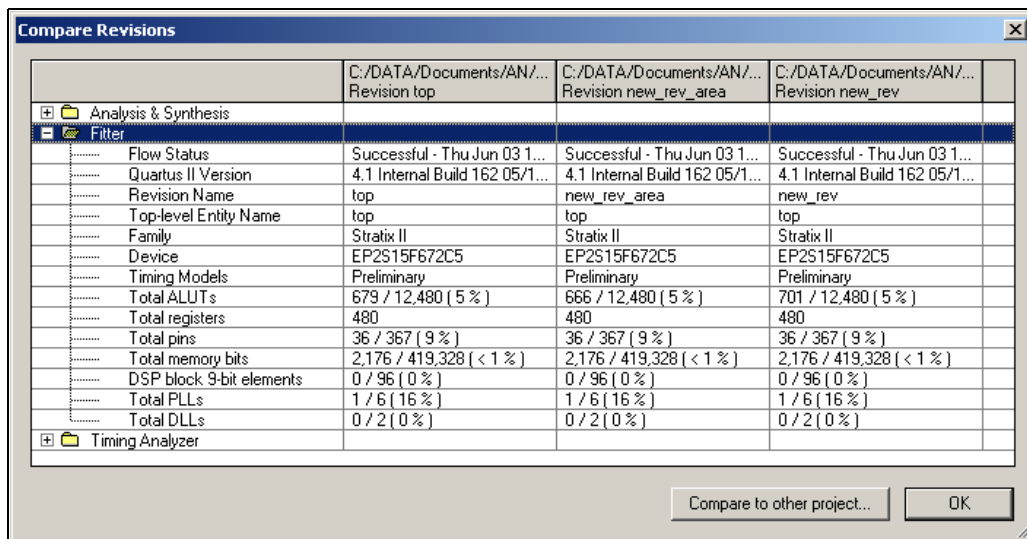


To delete the current revision, select a different revision as the current revision first.

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 4-1](#)), the results of each revision in three categories (Analysis & Synthesis, Fitter, and Timing Analyzer) are compared side by side.

Figure 4–1. Compare Revisions Dialog Box



You can also compare revisions from another project. To do this, click **Compare to other project** in the **Compare Revisions** dialog box and select a QPF to compare with.

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.

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 with the **Copy Project** command (Project menu), and make the necessary changes to your design files.

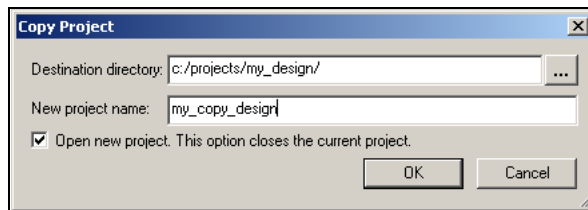
Creating a new version of your project with an Electronic Data Interchange Format (EDIF) or Verilog Quartus Mapping (VQM) netlist from a third-party EDA synthesis tool involves creating a copy of your project and then replacing the previous netlist file with the newly

generated netlist file. Use the **Copy Project** command (Project menu) to create a copy of your design and use the **Add/Remove Files from Project** command (Project menu) to add and remove design files.

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 4-2](#)).
2. **Browse** or type 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 in Quartus II** option.
5. Click **OK**.

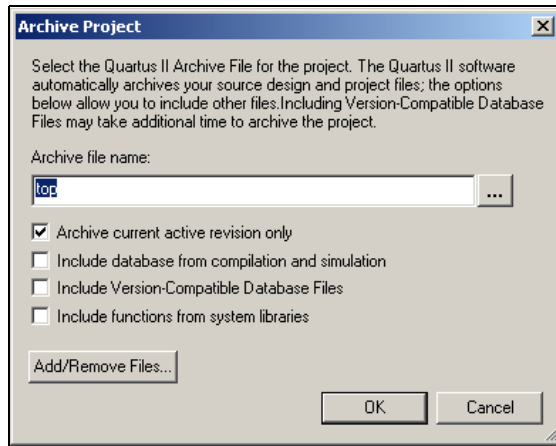
Figure 4-2. Copy Project Dialog Box



Archiving Projects

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. The archive file generated by the Archive Project feature (see [Figure 4-3](#)) can easily be shared between engineers.

Figure 4–3. Archive Project Dialog Box

To archive a project:

1. If you have not already done so, create a new project or open an existing project.
2. If you want, analyze or compile the design.
3. Choose **Archive Project** (Project menu).
4. Type a name for the Quartus II Archive File (.qar) in **Archive file name**, or select a name with **Browse (...)**.
5. To include the outputs of compilation and simulation, turn on **Include database from compilation and simulation**.
6. To include the Version-Compatible Database Files, turn on **Include Version-Compatible Database Files**.
7. To include functions from system libraries, turn on **Include functions from system libraries**.
8. Click **Add/Remove Files** to edit the contents of the QAR file.
9. Click **OK**.



Altera® recommends that you perform Analysis and Synthesis before archiving a project to ensure that all design files are located and archived.

To restore an archived project:

1. Choose **Restore Archived Project** (Project menu).
2. In the **Archive name** box, type the path and file name of the Quartus II Archive File (.qar) you wish to restore, or select a QAR File with **Browse (...)**.
3. In the **Destination folder** box, type or select the path of the folder into which you wish to restore the contents of the QAR File, or select a folder 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 necessary, recompile the project.

Version-Compatible Databases

In the past, 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 of the Quartus II software. For example, with the same 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 versions 4.1 and later without having to recompile your design.

Perform the following steps to export a version-compatible database:

1. Choose **Export Database** (Project menu).
2. **Browse** or type in a path in the **Export Directory** box.
3. Click **OK**.

Perform the following steps to import a version-compatible database:

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 **Save the database in a version-compatible format** option.
4. **Browse** to the directory in which you want to save the database.
5. Click **OK**.

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*.

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 and 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

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” on page 4–5](#).

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:

```
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 [page 4–7](#).

The following Tcl command restores the project archive named **archive.qar** in the subdirectory named **restored** and overwrites existing files:

```
project_restore archive.qar -destination restored -overwrite
```

Type the following command at a command prompt to restore a project archive:

```
quartus_sh --restore archive.qar ↵
```

Importing and Exporting Version-Compatible Databases

You can import and export version-compatible databases with either a Tcl command or a command run at a command prompt. For more information about importing and exporting version-compatible databases, see [“Version-Compatible Databases” on page 4–7](#).



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 will also need to 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>
```

Conclusion

Throughout the development of a successful FPGA design, designers often try different settings and versions of their designs. The **Revisions** feature in the Quartus II software facilitates the creation and management

of revisions, which are 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.



Section II. Device & Board Utilities

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	June 2004 v2.0	<ul style="list-style-type: none">● Scripting support section added.● Updated coding examples.
	Feb. 2004 v1.0	Initial release

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 draw limitations.

This chapter describes the design flow to assign and analyze pin-outs using the **Start I/O Assignment Analysis** command in the Quartus® II software, both before and after completion of your design.

I/O Assignment Planning & Analysis

Time-to-market constraints means that board layout is often done in parallel with, or even prior to, creating your design. Therefore, checking the legality of your I/O assignments early in the design process is often a requirement.

The **Start I/O Assignment Analysis** command in the Quartus II software provides the capability of checking your I/O assignments early in the process. You can use this command to check the legality of your pin assignments before, during, or after completion 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.

The **Start I/O Assignment Analysis** command is available for the Stratix® II, Stratix GX, Stratix, MAX® 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](#). This flow does not need 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 uses design files to thoroughly check the legality of your pin assignments and surrounding logic. 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.

Altera suggests that you 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.

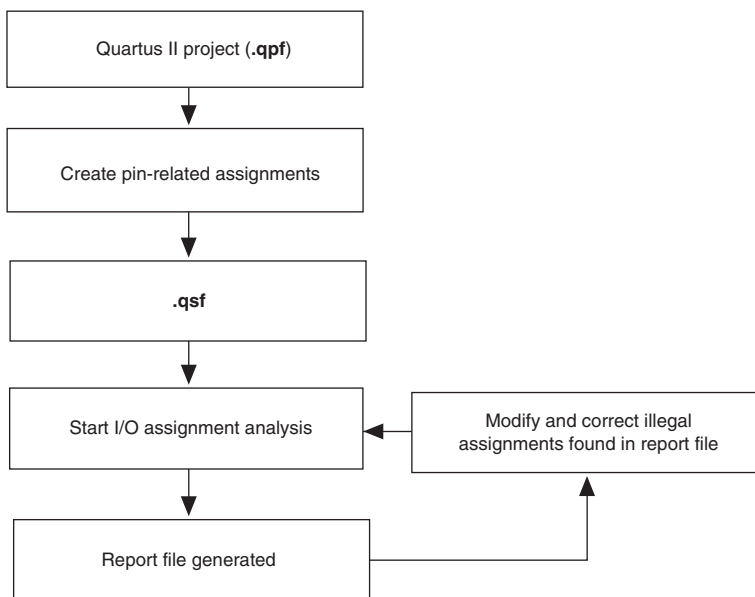
Along with the pin-related assignments, the **Start I/O Assignment Analysis** command also checks blocks that directly feed or are fed by a pin such as a phase-locked loop (PLL), low-voltage differential signal (LVDS), or gigabit transceiver block.

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.

Figure 5–1. Assigning & Analyzing Pin-outs without Design Files

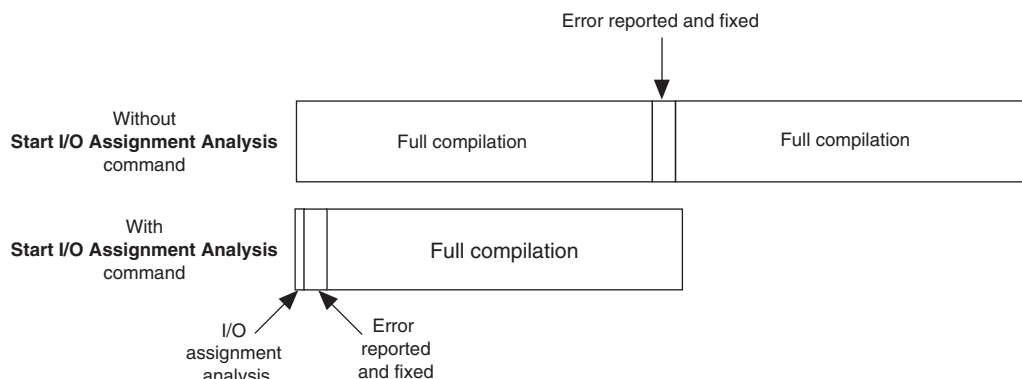
You can assign and analyze pin-outs using the **Start I/O Assignment Analysis** command without design files by following these steps:

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 [“Creating I/O Assignments” on page 5–6](#).
3. Choose **Start > Start I/O Assignment Analysis** (Processing menu) to start the analysis.
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 Complete or Partial 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](#) describes 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 can be checked by the I/O assignment analysis depends 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, the **Start I/O Assignment Analysis** command checks the legality of those pin-related assignments for which it has enough information.

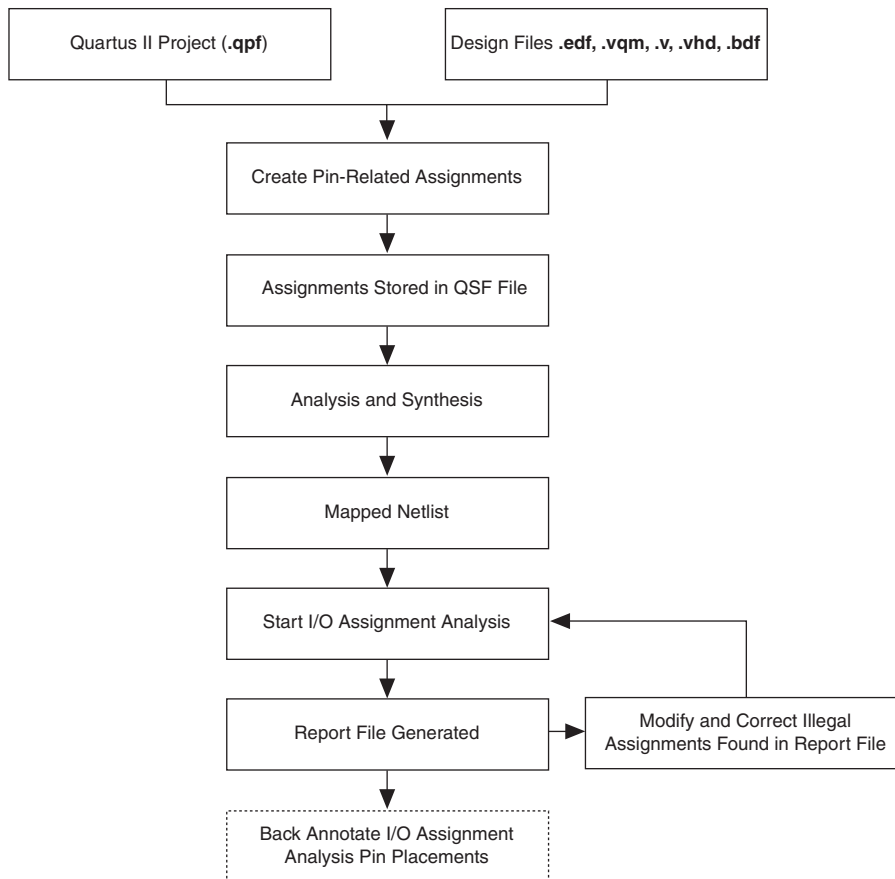
For example, you might assign a clock to a user I/O pin instead of assigning it to a dedicated clock pin. You design the clock to drive a PLL that has not yet been instantiated in the design. Because the **Start I/O Assignment Analysis** command is unaware of the logic that the pin drives, it is not able to check that only a dedicated clock input pin can drive the clock port of a PLL.

If you have a partial design, Altera recommends that you provide 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® files in your project for analysis.



A top-level wrapper file would be an example of a partial design.

Figure 5–3. Assigning & Analyzing Pin-outs with Design Files



Use the following steps to assign and analyze pin-outs using the **Start I/O Assignment Analysis** command with design files:

1. Create a Quartus II project and include your design files in the project.
2. Create pin-related assignments with the **Assignment Editor**.
3. Choose **Start > Start Analysis & Synthesis** (Processing menu) to generate an internal mapped netlist.

4. Choose **Start > Start I/O Assignment Analysis** (Processing menu) to start the analysis.
5. View the messages in the report file or in the **Messages** window
6. Correct any errors and violations reported
7. Rerun the **Start I/O Assignment Analysis** command until all errors are corrected.

Inputs Used 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 then the **Start I/O Assignment Analysis** command reads only the QSF.

If you have a partial or complete design, the **Start I/O Assignment Analysis** command reads in the QSF and the mapped netlist file.

Creating I/O Assignments

You can create pin-related assignments using the following features:

- Assign SignalProbe Pins dialog box
- Assignment Editor
- Tcl Commands
- Floorplan Editor

Reserving Pins

If you do not have any design files, you must create reserved pin assignments in addition to your other pin-related assignments. Reserving pins is necessary so that the **Start I/O Assignment Analysis** command will understand the pin type (input, output, or bidirectional) and correctly analyze the pin. You can reserve a pin by choosing **Assignment Editor** (Assignments menu), and selecting **Reserved Pin** from the Category list. In the spreadsheet interface, type in the pin name and select from the reserved list (see [Figure 5-4](#)).

Figure 5–4. Reserving a Pin with the Assignment Editor

	To	Reserved	
1	clk	As input tri-stated	
2	<<new>>	As bidirectional	
		As input tri-stated	
		As output driving an unspecified signal	
		As output driving ground	
		As SignalProbe output	



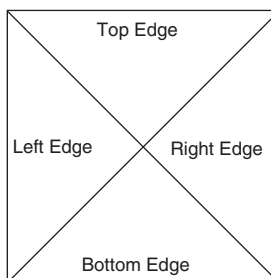
For more information on using the **Assignment Editor**, see the *Assignment Editor* chapter in Volume 2 of the *Quartus II Handbook*.

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. In the spreadsheet interface, type in the pin name and select a location from the location list. For Stratix II, Stratix GX, Stratix, and Cyclone devices, you can also assign a pin to an I/O Bank or Edge location.

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.

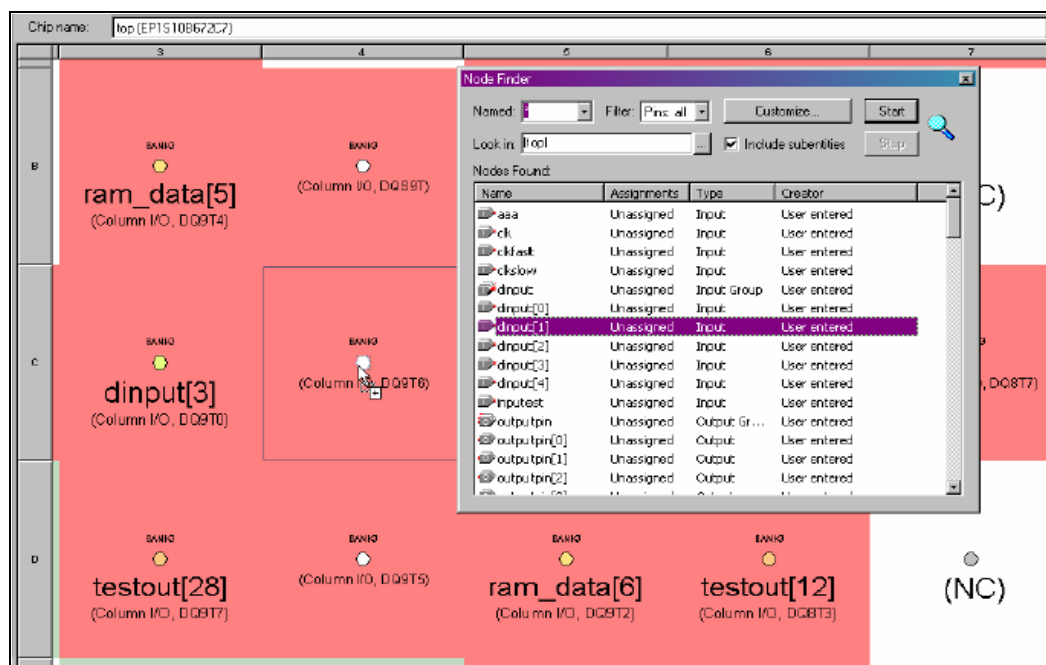
An easy way to place large buses that exceed the pins available in a particular I/O bank is to use Edge location assignments. You can also use Edge location assignments to improve circuit board routing ability of large buses, since they are close together near an edge. [Figure 5–5](#) shows the Altera device package edges.

Figure 5–5. Package View of the Four Edges on an Altera Device

Assignments with the Floorplan Editor

You can also make pin location assignments with the **Floorplan Editor**. Open the **Timing Closure Floorplan** by choosing **Timing Closure Floorplan** (Assignments menu). In the **Timing Closure Floorplan**, you can change the view between the package view and the interior cell view from the View menu. You can use the top and bottom package view to view the pins in the desired package. You can find the pad separation between two pins with the interior cell view. In both views, you can drag and drop pins from the Node Finder or from a graphic design file (GDF) or block design file (BDF) file into the desired pin or bank (see [Figure 5-6](#)).

Figure 5-6. Creating Pin Location Assignments with the Node Finder & the Timing Closure Floorplan



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.

Choose **Start > Start Analysis & Synthesis** (Processing menu) to generate a mapped netlist. You can also use the **quartus_map** executable to run analysis and synthesis.

The mapped netlist is stored internally in the Quartus II database.

Running the I/O Assignment Analysis

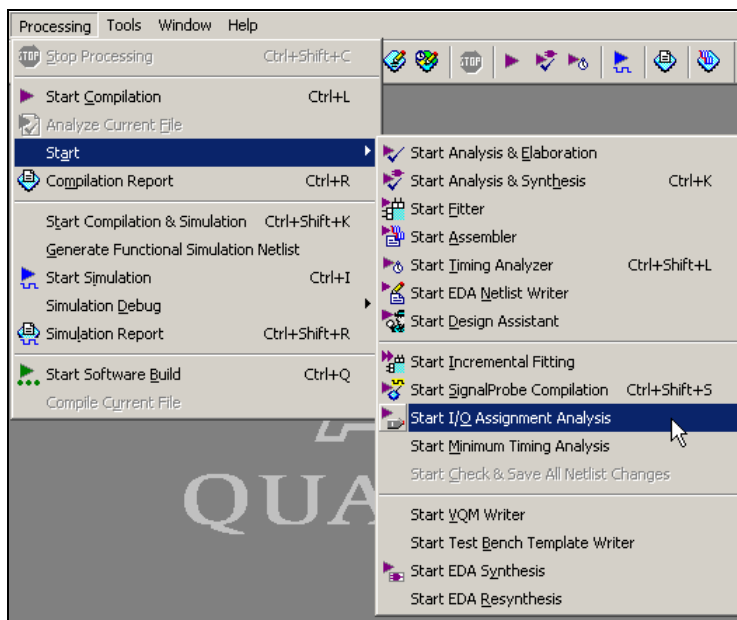
You can run the **Start I/O Assignment Analysis** command from the Quartus II software menu (see [Figure 5–7](#)) or from the command prompt. Choose **Start > Start I/O Assignment Analysis** (Processing menu) or type the following command in your project directory.

```
quartus_fit <project-name> --check_ios ←
```



Running the **Start I/O Assignment Analysis** command overwrites any previous fitter database. You can still view the previous compilation report text file.

Figure 5–7. I/O Assignment Analysis Command from the Quartus II Software Menu



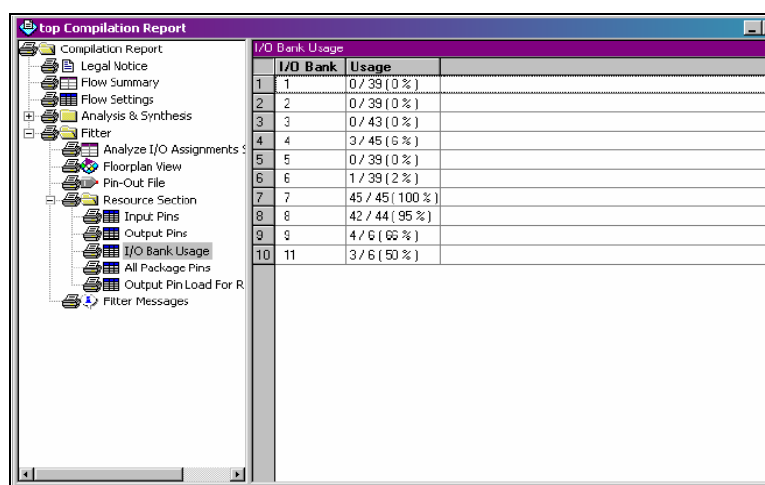
Understanding the I/O Assignment Analysis Report

The **Start I/O Assignment Analysis** command generates a detailed analysis report (see [Figure 5–8](#)) and a Pin-out File (.pin). You can view the report file by choosing **Compilation Report** (Project menu). The Fitter page of the Compilation report contains the following five sections:

- Analyze I/O Assignment Summary
- Floorplan View
- Pin-Out File
- Resource Section
- Fitter Messages

The Resource Section categorizes the pins as Input Pins, Output Pins, and Bidir Pins. You can use the **I/O Bank Usage** page under the **Resource Section** to view the utilization of each I/O bank in your device.

Figure 5–8. Summary of the I/O Bank Usage in the I/O Assignment Analysis Report



I/O Bank Usage		
	I/O Bank	Usage
1	1	0 / 39 (0 %)
2	2	0 / 39 (0 %)
3	3	0 / 43 (0 %)
4	4	3 / 45 (5 %)
5	5	0 / 39 (0 %)
6	6	1 / 39 (2 %)
7	7	45 / 45 (100 %)
8	8	42 / 44 (95 %)
9	9	4 / 6 (66 %)
10	11	3 / 6 (50 %)

The **Fitter Messages** page stores all messages including errors, warnings, and information messages. See [“Detailed Error/Status Messages” on page 5–11](#) for more information.

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.

Detailed Error/Status Messages

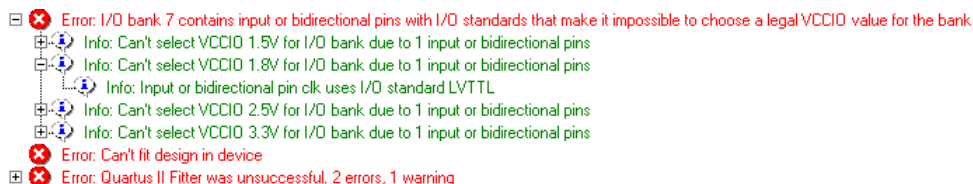
The **Start I/O Assignment Analysis** command provides detailed messages to help you quickly understand and resolve pin assignment errors. Each detailed message includes a related node name and a description of the problem.

You can view the 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**.

Following is an example of error messages reported by I/O assignment analysis:

Figure 5–9. Error Message Report by I/O Assignment Analysis

- 
- ✖ Error: I/O bank 7 contains input or bidirectional pins with I/O standards that make it impossible to choose a legal VCCIO value for the bank
 - ⊕ Info: Can't select VCCIO 1.5V for I/O bank due to 1 input or bidirectional pins
 - ⊕ Info: Can't select VCCIO 1.8V for I/O bank due to 1 input or bidirectional pins
 - ⊕ Info: Input or bidirectional pin clk uses I/O standard LVTTTL
 - ⊕ Info: Can't select VCCIO 2.5V for I/O bank due to 1 input or bidirectional pins
 - ⊕ Info: Can't select VCCIO 3.3V for I/O bank due to 1 input or bidirectional pins
 - ✖ Error: Can't fit design in device
 - ✖ Error: Quartus II Fitter was unsuccessful. 2 errors, 1 warning

Scripting Support

You can run procedures and make settings described in this chapter with a 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 this handbook. For more information about command-line scripting, see the *Command-Line Scripting* chapter in Volume 2 of this handbook. For detailed information about scripting command options type `quartus_sh --qhelp` at a system command prompt.

Reserving Pins

Use the following Tcl command to reserve a pin. For more information about reserving pins, see [page 5–6](#).

```
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 [page 5–7](#).

```
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 5–8](#).

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>↵
```

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 5–9](#).

Enter the following in a Tcl console or script:

```
execute_flow -check_ios
```

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 and Physical Synthesis](#)
- [Chapter 9, Design Space Explorer](#)
- [Chapter 10, LogicLock Design Methodology](#)
- [Chapter 11, Timing Closure in HardCopy Devices](#)
- [Chapter 12, Synplicity Amplify Physical Synthesis Support](#)

Revision History

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

Chapter(s)	Date / Version	Changes Made
6	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	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
8	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
9	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
10	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, figures. • New functionality in the Quartus II software version 4.1.
	Feb. 2004 v1.0	Initial release
11	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
12	Feb. 2004 v1.0	Initial release.

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.



For more information on a block-based design approach, see the *Hierarchical Block-Based & Team-Based Design Flows* chapter in Volume 1 of the *Quartus II Handbook*.

This chapter explains techniques to reduce resource usage, improve timing performance, and reduce compile 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. Quartus II options and settings 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.

The first stage in the optimization process is to perform an initial compilation (see [“Initial Compilation” on page 6–2](#)) to establish a baseline that you can use to analyze your design. [“Design Analysis” on page 6–6](#) explains how to analyze the results of your design, and provides links to the sections of this chapter where you can proceed with resource or performance optimization. Altera recommends optimizing resource usage first, then I/O timing, then f_{MAX} timing, so this chapter presents the recommendations for each stage in the appropriate order. This chapter first documents this analysis and optimization process for look-up table (LUT)-based devices, including FPGA devices and MAX® II device family

CPLDs. It then focuses on the process for MAX 7000 and MAX 3000 device family macrocell-based CPLDs. The final optimization section covers compilation time optimization, which is device independent.

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 the Quartus II software can use.

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 6–4](#)). The Quartus II physical synthesis optimizations are also performed based on the constraints in specified timing requirements (see [“Synthesis Netlist Optimizations and Physical Synthesis Optimizations” on page 6–28](#) 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 which paths violate the timing requirements.

Make timing requirement settings in the **Timing Requirement & Options** page of the **Settings** dialog box (Assignments menu) or with the Assignment Editor. On the **Timing Requirement & 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 makes it easy to make individual clock settings.

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 any 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 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 any timing constraints in the design, the Quartus II software does not attempt to optimize clocks that are unconstrained. Specify timing constraints on all clock signals in the design wherever possible 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 compile 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** page of the **Settings** dialog box (Assignments menu).

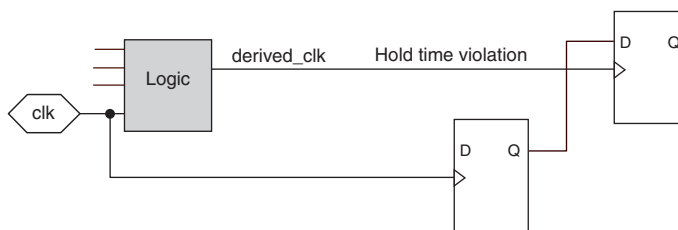
Timing Driven Compilation Settings

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. If you choose **I/O Paths and Minimum TPD Paths** (the default choice), 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 6–1](#), where a derived clock generated with logic causes a hold time problem on another register. However, if your design has internal hold time violations between registers, 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 6–1. 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 modify 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 unroutable, resulting in multiple fitting attempts and a long fitting time.

The **Auto Fit** option (available for Stratix II, Stratix, Stratix GX, Cyclone II, Cyclone, and MAX II devices only) 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. This option also causes the Quartus II Fitter to optimize for shorter compile 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 compile time is the same as using the **Standard Fit** option. For designs with easy or no timing requirements, the **Auto Fit** option reduces compile time by 40% on average.



Note that selecting this option does not guarantee that the Fitter will meet 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. [“Optimization Techniques for Macrocell-Based \(MAX 7000 and MAX 3000\) CPLDs”](#) on page 6–41 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) or Tcl script commands.



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*. For information on scripting, see the *Tcl Scripting* 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) see “[Optimization Techniques for LUT-Based \(FPGA and MAX II\) Devices](#)” on page 6–12:

- If your design does not fit, see “[Resource Utilization Optimization Techniques \(LUT-Based Devices\)](#)” on page 6–13 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 6–21 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 6–27.

For Macrocell-based devices (MAX 7000 and MAX 3000 CPLDs) see “[Optimization Techniques for Macrocell-Based \(MAX 7000 and MAX 3000\) CPLDs](#)” on page 6–41:

- If your design does not fit, see “[Resource Utilization Optimization Techniques \(Macrocell-based CPLDs\)](#)” on page 6–41 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 6–49.

For techniques to reduce the compilation time, see “[Compilation Time Optimization Techniques](#)” on page 6–55.

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 spreads out a design as much as possible while trying to meet any timing constraints set by the user. 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 6-2. 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 when the device runs in your system.

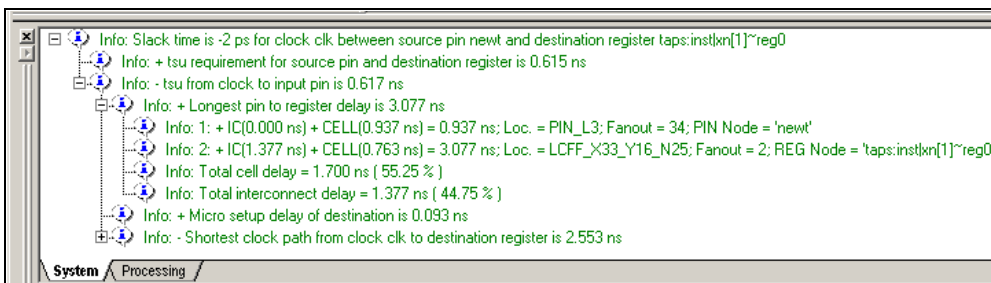
Figure 6-2. I/O Timing Report

	Slack	Required tsu	Actual tsu	From	To	To Clock
1	-0.002 ns	0.615 ns	0.617 ns	newt	taps:instlkn[0]~reg0	clk
2	-0.002 ns	0.615 ns	0.617 ns	newt	taps:instlkn_2[1]~reg0	clk
3	-0.002 ns	0.615 ns	0.617 ns	newt	taps:instlkn_1[1]~reg0	clk
4	-0.002 ns	0.615 ns	0.617 ns	newt	taps:instlkn_1[1]~reg0	clk
5	-0.002 ns	0.615 ns	0.617 ns	newt	taps:instlkn_1[1]~reg0	clk
6	-0.002 ns	0.615 ns	0.617 ns	newt	taps:instlkn_1[1]~reg0	clk
7	0.000 ns	0.615 ns	0.615 ns	newt	taps:instlkn_1[1]~reg0	clk
8	0.000 ns	0.615 ns	0.615 ns	newt	taps:instlkn_1[1]~reg0	clk
9	0.000 ns	0.615 ns	0.615 ns	newt	taps:instlkn_1[1]~reg0	clk
10	0.000 ns	0.615 ns	0.615 ns	newt	taps:instlkn_1[1]~reg0	clk
11	0.000 ns	0.615 ns	0.615 ns	newt	taps:instlkn_1[1]~reg0	clk
12	0.000 ns	0.615 ns	0.615 ns	newt	taps:instlkn_1[1]~reg0	clk
13	0.000 ns	0.615 ns	0.615 ns	newt	taps:instlkn_1[1]~reg0	clk
14	0.000 ns	0.615 ns	0.615 ns	newt	taps:instlkn_1[1]~reg0	clk
15	0.000 ns	0.615 ns	0.615 ns	newt	taps:instlkn_1[1]~reg0	clk
16	0.076 ns	0.615 ns	0.539 ns	newt	taps:instlkn_1[2]~reg0	clk
17	0.076 ns	0.615 ns	0.539 ns	newt	taps:instlkn_3[2]~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 6-2). A message listing the paths appears in the **System** tab of the **Messages** window. You can expand a selection by clicking the “+” icon at the beginning of the line, as shown in Figure 6-3. 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 6–3. 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** (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 (i.e., 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

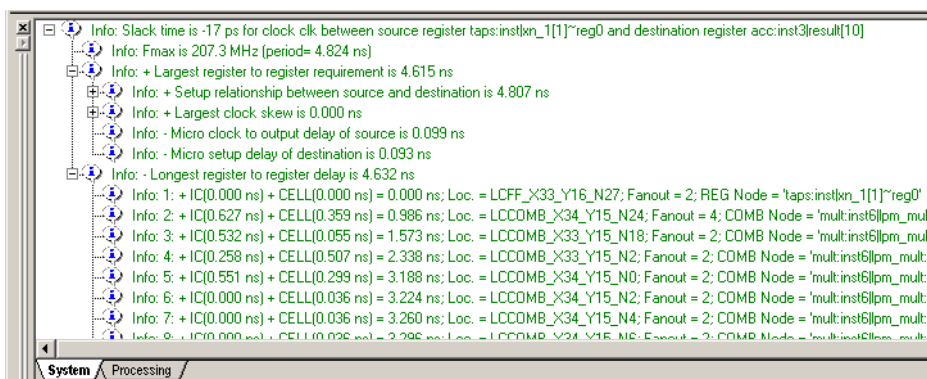
To determine whether f_{MAX} timing requirements are met, see the **Timing Analyzer** section of the **Compilation Report** (Processing menu). The **Clock Setup** folder gives figures for the actual register-to-register f_{MAX}, as reported by the Quartus II software, and the slack, or difference between 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 6–4).

Figure 6-4. t_{MAX} Timing Analysis Report

Clock Setup: 'clk'			
	Slack	Actual fmax (period)	
1	-0.081 ns	204.58 MHz (period = 4.888 ns)	taps:inst[kn[1]]~reg0
2	-0.053 ns	205.76 MHz (period = 4.860 ns)	taps:inst[kn_1[1]]~reg0
3	-0.045 ns	206.10 MHz (period = 4.852 ns)	taps:inst[kn[1]]~reg0
4	-0.017 ns	207.30 MHz (period = 4.824 ns)	
5	-0.009 ns	207.64 MHz (period = 4.816 ns)	
6	0.005 ns	208.25 MHz (period = 4.802 ns)	
7	0.019 ns	208.86 MHz (period = 4.788 ns)	
8	0.027 ns	209.21 MHz (period = 4.780 ns)	
9	0.041 ns	209.82 MHz (period = 4.766 ns)	
10	0.041 ns	209.82 MHz (period = 4.766 ns)	
11	0.055 ns	210.44 MHz (period = 4.752 ns)	
12	0.063 ns	210.79 MHz (period = 4.744 ns)	
13	0.077 ns	211.42 MHz (period = 4.730 ns)	
14	0.077 ns	211.42 MHz (period = 4.730 ns)	
15	0.099 ns	212.40 MHz (period = 4.708 ns)	
16	0.100 ns	212.45 MHz (period = 4.707 ns)	state_minst1filter~30

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 6-4). A message listing the paths appears in the **System** tab of the Messages window. You can expand a selection by clicking the “+” icon at the beginning of the line, as shown in Figure 6-5. 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 6–5. t_{MAX} Slack Report

You can visually analyze t_{MAX} paths by right-clicking on a path in the report and selecting **Locate in Timing Closure Floorplan** to display the **Timing Closure Floorplan**, which then highlights the path. You can also view all failing paths in the **Timing Closure Floorplan** using 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*.

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.

Optimization Techniques for LUT-Based (FPGA and MAX II) Devices

This section of the chapter addresses resource and timing optimization issues for LUT-based Altera devices, which consists of all FPGA devices and MAX II device family CPLDs.

For information on optimizing MAX 7000 and MAX 3000 CPLD designs, refer to [“Optimization Techniques for Macrocell-Based \(MAX 7000 and MAX 3000\) CPLDs” on page 6–41](#). For information on optimizing compilation time (when targeting any device), refer to [“Compilation Time Optimization Techniques” on page 6–55](#).

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.

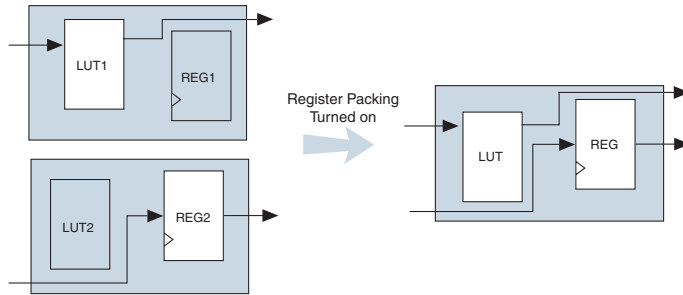
Resource Utilization Optimization Techniques (LUT-Based Devices)

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 6–2. If a design is not fitting into a specified device, use the techniques in this section to achieve a successful fit.

Use Register Packing

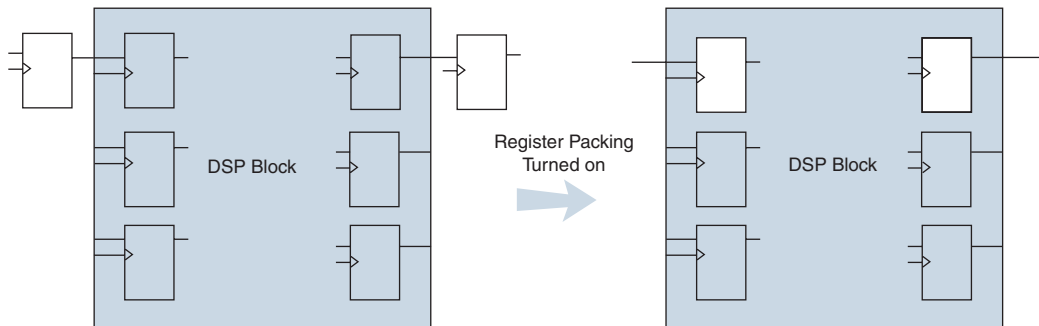
The **Auto Packed Registers** option is available regardless of the tool used to synthesize the design. Register packing combines a logic cell where only the register is used with another logic cell where only the lookup table (LUT) is used, and implements both functions in a single logic cell. [Figure 6–6](#) shows the packing and the gain of one logic cell.

Figure 6–6. Register Packing



Registers may also be packed into DSP blocks as shown in [Figure 6–7](#).

Figure 6–7. 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. 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 6–1](#) shows typical results for Stratix II devices, [Table 6–2](#) shows typical results for Cyclone II devices, and [Table 6–3](#) 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 6–1. 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 6–2. 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 6–3. Typical Register Packing Results for Stratix, Stratix GX, and Cyclone Devices

Register Packing Setting	Relative f_{MAX}	Relative LE Count
Off	1.00	1.12
Normal	1.00	1.00
Minimize Area	0.97	0.93
Minimize Area with Chains	0.94	0.90
Auto (default)	1.0 until device is very full, then gradually to 0.94 as required	1.0 until device is very full, then gradually to 0.90 as required

Remove Fitter Constraints

A design with too many user constraints may not fit the targeted device. This occurs 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 and/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.



For more information on the **Routing Congestion** view in the **Timing Closure Floorplan**, see the Quartus II Help.

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

If your design fails to fit because it uses too much logic, resynthesize the design to improve the area utilization, as follows.

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.

Optimize for Area, Not Speed

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} timing performance) while leaving the default **Optimization Technique** setting at **Balanced** (for the best trade-off between area and speed for certain device families) or **Speed**. 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

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.

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 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 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 and edit the block so it targets logic instead of DSP blocks.

DSP blocks can 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.

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 also 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.

Resolving Resource Utilization Issues Summary

Table 6–4 shows design options used to reduce excess 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 Quartus II software includes the Design Space Explorer (DSE) Tcl/Tk script for automating successive compilations of a design, each employing different design options.



For more information on the DSE script, see the *Design Space Explorer* chapter in Volume 2 of the *Quartus II Handbook*.

Table 6–4. Techniques for Resolving Resource Utilization Issues (Part 1 of 2)

Issue	Design Options to Employ (in Order from Left to Right)					
Too many logic cells used or logic cells do not fit	Use register packing	Remove Fitter constraints	Perform WYSIWYG Primitive Resynthesis	Optimize synthesis for area / change state machine encoding	Optimize source code	Use a larger device
Too many memory blocks used	Retarget memory blocks	Modify synthesis options	Remove Fitter constraints	Optimize source code	Use a larger device	

Table 6–4. Techniques for Resolving Resource Utilization Issues (Part 2 of 2)

Issue	Design Options to Employ (in Order from Left to Right)					
Too many DSP blocks used	Retarget DSP blocks	Modify synthesis options	Remove Fitter constraints	Optimize source code	Use a larger device	
Problems placing I/O pins	Change pin assignments	Use a larger package with the same device density	Use a larger device with a larger pin count			
Too many routing resources used	Remove Fitter constraints	Modify synthesis options	Optimize source code	Use a larger device		

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.

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 6–2, 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.

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 load port, and the value is not **1** (Stratix, Stratix GX, and Cyclone devices only)
- Use the synchronous load or asynchronous clear port (APEX and APEX II devices only)

Registers with the above characteristics are optimized using the regular Quartus II Fitter optimizations.

Fast Input, Output, & Output Enable Registers

You can manually place individual registers in I/O cells 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. [Figure 6–8](#) shows the Resource Property Editor window displaying a programmable delay implemented in the delay chain of a Stratix device. 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 write changes directly to the netlist.



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

Figure 6–8. Delay Chain Shown in the Quartus II Resource Property Editor

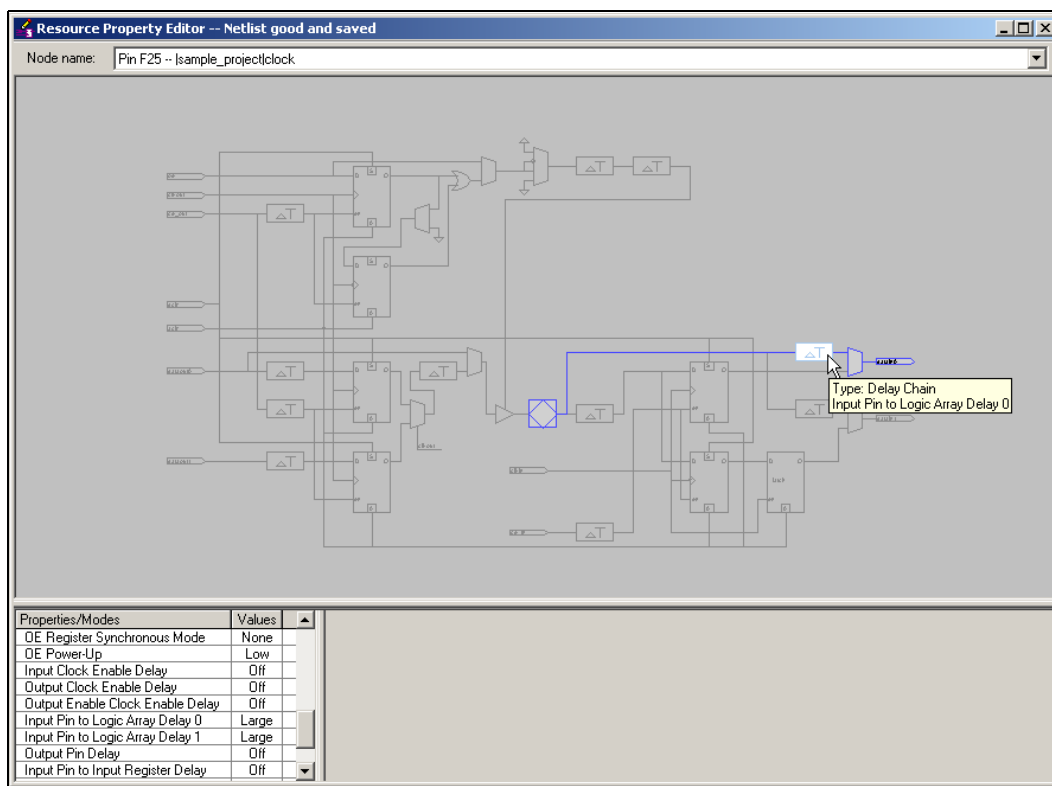


Table 6–5 summarizes the programmable delays available for Altera devices.

Table 6–5. Programmable Delays for Altera Devices (Part 1 of 3)

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

Table 6–5. Programmable Delays for Altera Devices (Part 2 of 3)

Programmable Delay	Description	I/O Timing Impact	Device Families
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
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

Table 6–5. Programmable Delays for Altera Devices (Part 3 of 3)

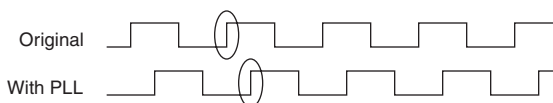
Programmable Delay	Description	I/O Timing Impact	Device Families
Increase output clock enable delay	Increases the propagation delay from the interior of the device to the clock enable input of the I/O output register and output enable register.	N/A	Stratix, Stratix GX, APEX II, APEX 20KE, APEX 20KC
Increase 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

Using 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.

Using 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 . This technique can be used only in devices that offer PLLs with the phase shift option. See [Figure 6–9](#).

Figure 6–9. Shift Clock Edges Forward to Improve t_{SU} at the Expense of t_{CO} 

Improving Setup & Clock-to-Output Times Summary

Table 6–6 shows the recommended order in which to use techniques to reduce t_{SU} and t_{CO} times. Keep in mind that reducing t_{SU} times increases hold (t_H) times.

<i>Table 6–6. Improving Setup & Clock-to-Output Times</i> <i>Note (1)</i>		
Technique	t_{SU}	t_{CO}
Ensure that the appropriate constraints are set for the failing I/Os	✓	✓
Use timing-driven compilation for I/O	✓	✓
Use fast input register	✓	
Use fast output register and fast output enable register		✓
Set Decrease Input Delays to Input Register = ON or decrease the value of Input Delay from Pin to Input Register	✓	
Set Decrease Input Delays to Internal Cells = ON or decrease the value of Input Delay from Pin to Internal Cells	✓	
Set Increase Delay to Output Pin = OFF or decrease the value of Delay from Output Register to Output Pin		✓
Use PLLs to shift clock edges	✓	✓
Use the Fast Regional Clock option		✓

Note to Table 6–6:

(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.

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.

Synthesis Netlist Optimizations and 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 standard 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 Re-timing**

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 also specify the **Physical synthesis effort**, which sets the level of physical synthesis optimization you want the Quartus II software to perform. You can specify the physical synthesis optimization options on the **Physical Synthesis Optimizations** page under **Fitter Settings** in the **Settings dialog** box (Assignments menu).



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, use these options in different combinations. Performance results are design dependant. Typical benchmark results with netlists from a leading third-party synthesis tool and compiled with the Quartus II software version 4.1 are shown in Table 6-7. These results were obtained for Stratix devices, using various designs and numbers of LEs.

Table 6-7. Average Performance of Different Netlist Optimizations

Optimization Method	f _{MAX} Gain (%)	Win Ratio (%) (1)	Winner's f _{MAX} Gain (%) (2)	Change Logic (%)	Increase in Compile Time (x)
WYSIWYG primitive resynthesis	2	60	6	-8	1.0
Physical synthesis for combinational logic and registers					
Using physical synthesis Fast effort level	10	86	14	4	1.7
Using physical synthesis Normal effort level	15	86	14	4	2.7
Using physical synthesis Extra effort level	17	86	14	4	4.2
WYSIWYG primitive re-synthesis as well as physical synthesis for combinational logic and registers					
Using physical synthesis Fast effort level	12	87	16	-5	1.7
Using physical synthesis Normal effort level	17	87	16	-5	2.7
Using physical synthesis Extra effort level	19	87	16	-5	4.2
All options on (WYSIWYG primitive re-synthesis, gate level register re-timing, and physical synthesis for combinational logic and registers)					
Using physical synthesis Extra effort level	19	82	17	-6	4.3

Notes to Table 6-7:

- (1) Win is the percentage of designs that showed better performance with the option on, than without the option on.
- (2) Winner's f_{MAX} gain refers to the average improvement for the designs that showed better performance with these settings (designs considered a Win).

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.

The DSE Tcl/Tk script can automate successive compilations of a design, each employing different netlist optimization options.



For more information on the DSE script, see the *Design Space Explorer* chapter in Volume 2 of the *Quartus II Handbook*.

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 from seed to seed changes.

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.



For more information on the DSE script, see the *Design Space Explorer* chapter in Volume 2 of the *Quartus II Handbook*.

Optimize Synthesis for Speed

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, use the recommendations in the following paragraphs.



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.

You can use the DSE to experiment with different Quartus II synthesis options to optimize for best performance.



For more information, see the *Design Space Explorer* chapter in Volume 2 of the *Quartus II Handbook*.

Optimize for Speed, Not Area

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

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.

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. For more information on the **Manual Logic Duplication** option, see the Quartus II Help.

Other Synthesis Options

With your synthesis tool, experiment with the following options if they are available:

- Register balancing or retiming
- Register pipelining

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 recoding 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 on the device will help improve the routing delay.



Note that improving fitting results, especially for larger devices such as Stratix and Stratix II, can be difficult. 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.

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 over-constraint. The following types of LogicLock assignments are available, listed in order of decreasing flexibility:

- Soft LogicLock regions
- Auto size, floating location regions
- Fixed size, floating location regions
- Fixed size, locked location regions

To determine what to put into a LogicLock region, 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 6–8](#), mod_A is probably a problem module. In this case, mod_A could be placed in a LogicLock region to attempt to put all the nodes in the module closer together in the floorplan.

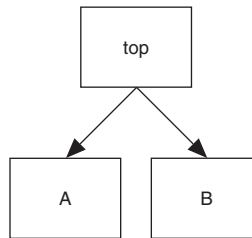
Figure 6–10. Design Hierarchy

Table 6–8 shows the failing module paths in timing analysis.

Table 6–8. Failing Module Paths in Timing Analysis

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

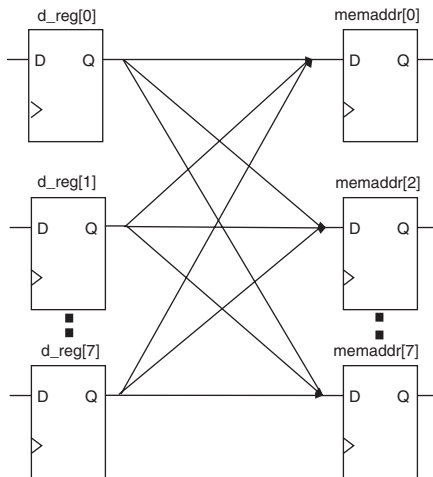
Path Assignments

If you see a pattern such as the one shown in Figure 6–11 and Table 6–9, 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 6–9. 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 than the Quartus II Fitter.



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.

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 overlaps with a custom region, such as part of a M-RAM, 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 with other logic blocks in the device.



For more information on a block-based design approach, see the *Hierarchical Block-Based Design & Team-Based Design Flows* chapter in Volume 1 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.

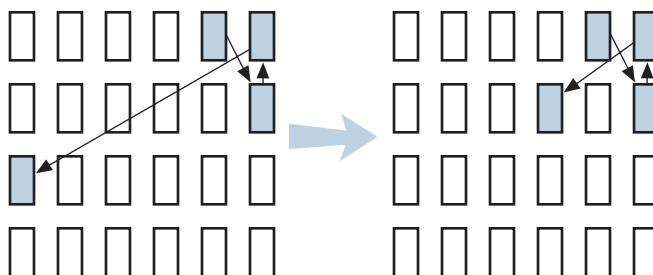


For more information on back-annotation of routing, see Quartus II Help.

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.

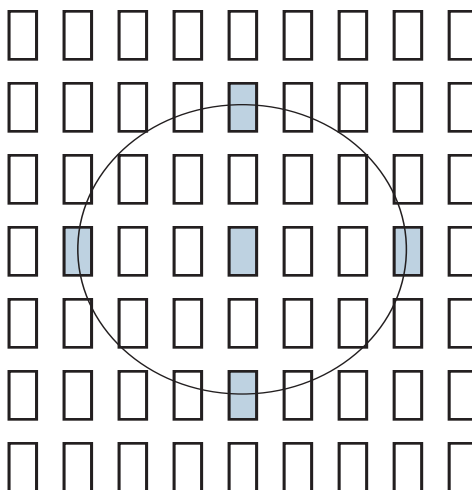
In general, when you are performing manual place-and-route, it is best to fix all I/O paths first. This is because there are often fewer options available to meet I/O timing. After I/O timing has been met, focus on manually placing f_{MAX} paths. This strategy follows the methodology outlined in this 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 rearchitect 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 of optimization than using LogicLock or location assignments.

If your critical path involved 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 allow 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*.

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).
2. Modify the seed. (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.
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.
6. Make location assignments, or perform manual placement by back-annotating the design.

If these options do not achieve performance requirements, design source code modifications may be required.



For more information on the DSE script, see the *Design Space Explorer* chapter in Volume 2 of the *Quartus II Handbook*.

Optimization Techniques for Macrocell-Based (MAX 7000 and MAX 3000) CPLDs

This section of the chapter addresses resource and timing optimization issues for Macrocell-based Altera devices, MAX 7000 and MAX 3000 CPLD device families.

For information on optimizing FPGA and MAX II CPLD designs, refer to [“Optimization Techniques for LUT-Based \(FPGA and MAX II\) Devices” on page 6–12](#). For information on optimizing compilation time (when targeting any device), refer to [“Compilation Time Optimization Techniques” on page 6–55](#).

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 6–2](#). 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 for primary logic can be preserved. 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 (e.g., 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 (i.e., 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 the control signals to dedicated input pins in one of four ways:

- In the Assignment Editor, choose one of two methods:
 - Assign pins to dedicated pin locations
 - Assign global signal settings 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 in the MAX+PLUS II software for the design, 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 recompilation 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 automatically choose pin assignments 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. Quartus II Help EPM3032A Dedicated Pin-Out Information

Dedicated Input Pins				Pin Numbers			
Function	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 (e.g., EPM3032A) in Quartus II Help to bring up the device pin table with output enable information.

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. Quartus II Help EPM3512A Dedicated Pin-Out Information

This column lists the possible sources for the output enable signals (i.e., GOE1, GOE2, etc), (1)

Function	Config. Pin <i>Note (10)</i>	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	—	—	E7

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.

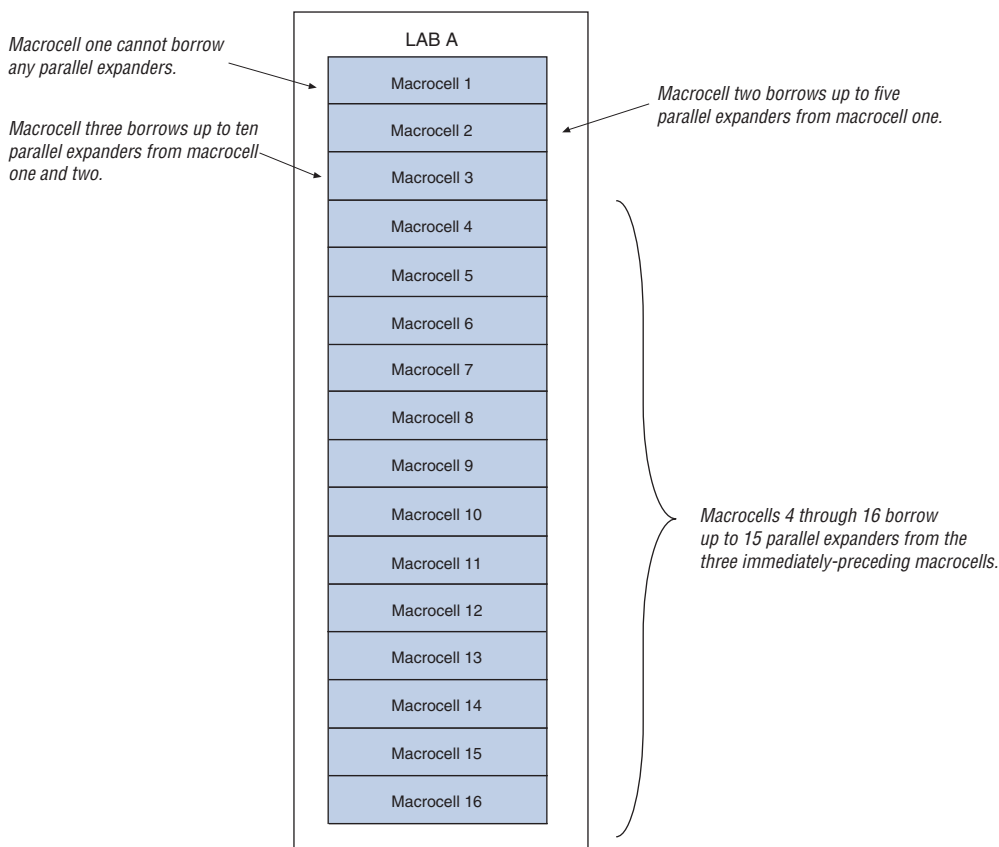
Estimate Fan-In When Assigning Output Pins

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 (e.g., 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

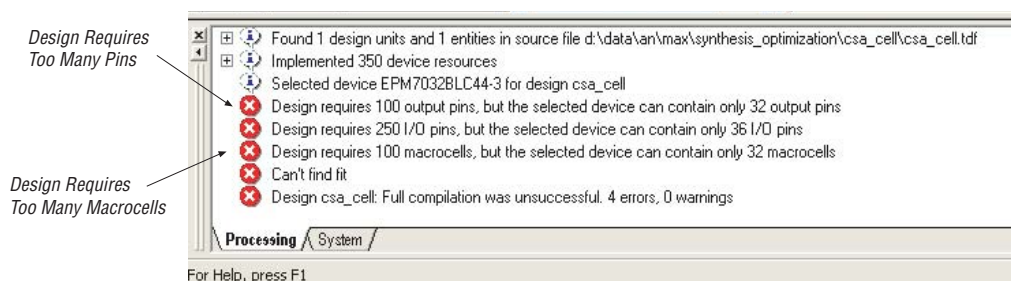
During compilation with the Quartus II software, you may receive an error message (see [Figure 6–17](#)) alerting you that the compilation was not successful.

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**—An algorithm that is written to give preference to device-fitting rather than device-speed (f_{MAX}) is selected when the Area Optimization technique is enabled. As expected, the device-fitting algorithm produces a slower compilation result. 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 6–46.
- Reduce the fan-in per cell—If you are not limited by the number of macrocells used in the design, you can use the **Fanin 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 will have more fitting flexibility for each macrocell, i.e., allowing macrocells to relocate. For example, each macrocell (previously grouped together in the same LAB) may move 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 6–48](#) 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 6–42](#).

Using LCELL Buffers to Reduce Required Resources

Complex logic, such as multi-level XOR gates, will often be 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.

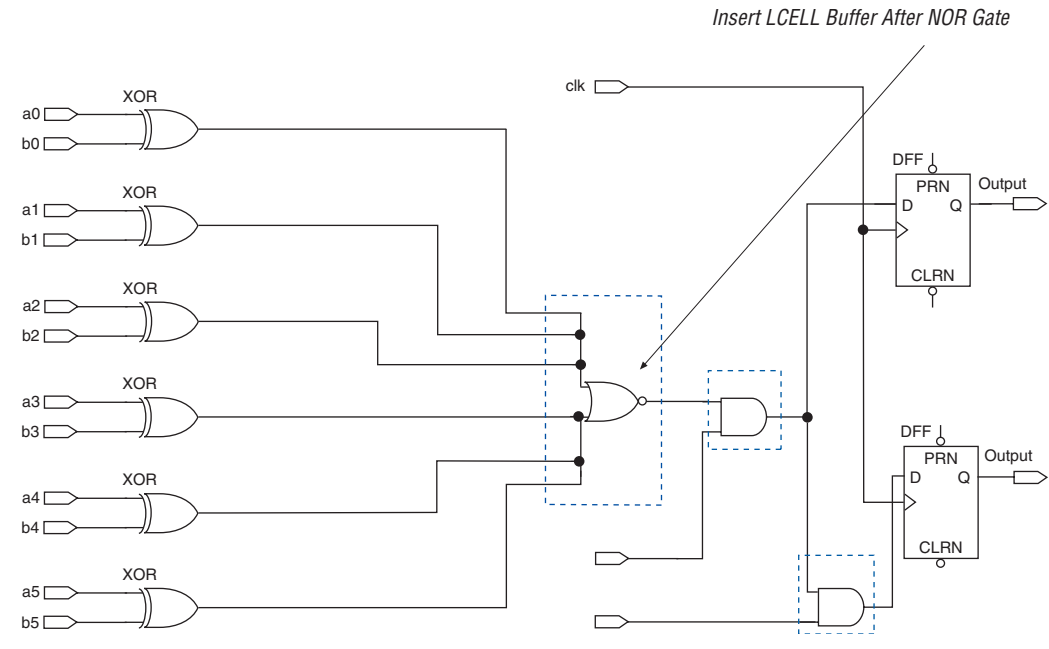
Instead of using the Quartus II software’s **Auto Logic Cell Insertion** option, you can manually insert logic cells. However, Altera recommends that you use the **Auto Logic Cell Insertion** option unless you know which part of the design is causing the congestion.

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



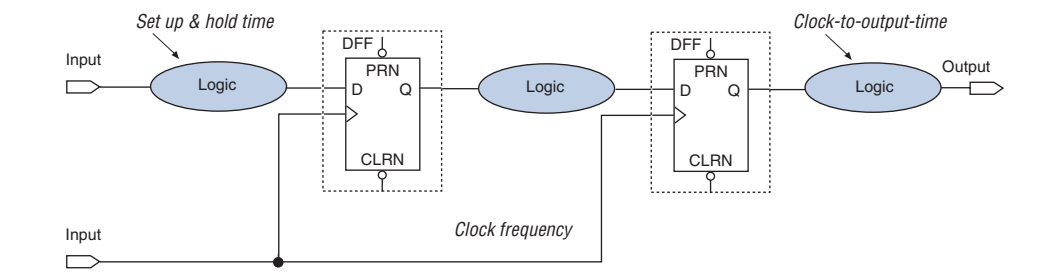
Timing Optimization Techniques (Macrocell-based CPLDs)

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 6–2, 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 6–6.

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 out to 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 guidelines listed below:

- Turn on the **Fast Input Register** option—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 guidelines listed below:

- 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 6–10 lists 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 6–10. EPM7064AETC100-4 Device Timing Results (Part 1 of 2)

Number of Registers	t_{SU}	t_H	t_{CO}	Global Clock Used	Fast Input Register Option	D Input Location	Q Output Location	Additional Logic Between D Input Location & Register	Additional Logic Between Register & Q Output Location
One	1.3 ns	1.2 ns	4.3 ns	No	On	LAB A	LAB A	No	No
One	1.6 ns	0.3 ns	4.3 ns	No	Off	LAB A	LAB A	No	No
One	2.5 ns	0 ns	3.1 ns	Yes	On	LAB A	LAB A	No	No

Table 6–10. EPM7064AETC100-4 Device Timing Results (Part 2 of 2)

One	2.8 ns	0 ns	3.1 ns	Yes	Off	LAB A	LAB A	No	No
One	3.6 ns	0 ns	3.1 ns	Yes	Off	LAB A	LAB A	Yes	No
One	2.8 ns	0 ns	7.0 ns	Yes	Off	LAB D	LAB A	No	Yes
16 registers with the same D and clock inputs	2.8 ns	0 ns	All 6.2 ns	Yes	Off	LAB D	LAB A, B	No	No
32 registers with the same D and clock inputs	2.8 ns	0 ns	All 6.4 ns	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 subdesigns can increase the performance of complex logic functions. However, if the project's pin or logic cell assignments use parallel expanders placed physically together with macrocells (which can reduce routability), parallel expanders can cause the Quartus II 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 on turning the **Auto Parallel Expanders** option on, refer to [“Resolving Macrocell Usage Issues” on page 6–46](#).
- 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 input(s) of the register(s) it feeds must be less than 10 ns. Maintaining the system clock speed can be difficult if there are long delay paths through complex logic. Altera recommends that you follow the guidelines below to improve your design's clock speed (i.e., 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 subdesigns can increase the performance of complex logic functions. However, if the project's pin or logic cell assignments use parallel expanders placed physically together with macrocells (which can reduce routability), parallel expanders can cause the Quartus II Compiler to have difficulties finding and optimizing a fit. Additionally, the 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 6–46](#).
- 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 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).
- 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 the 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 (i.e., out1, out2, out3, and out4) to the D-flip-flops in 4REG.

Figure 6–20. A 2- to 4-Pipelined Decoder

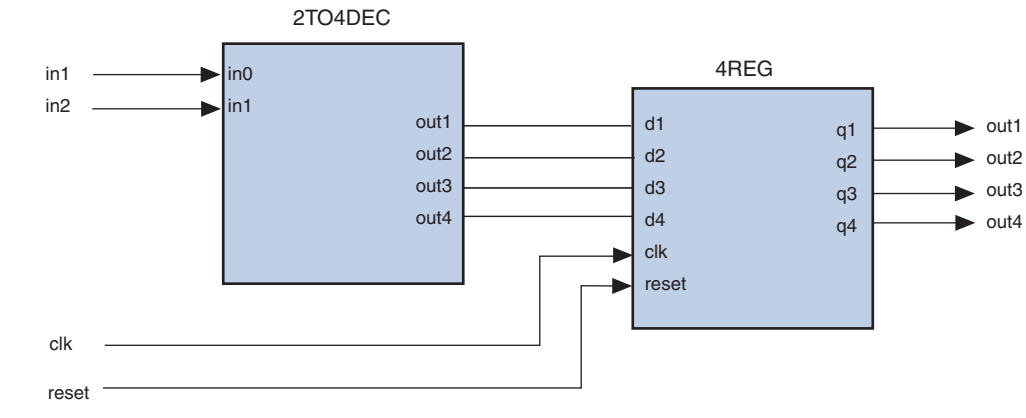
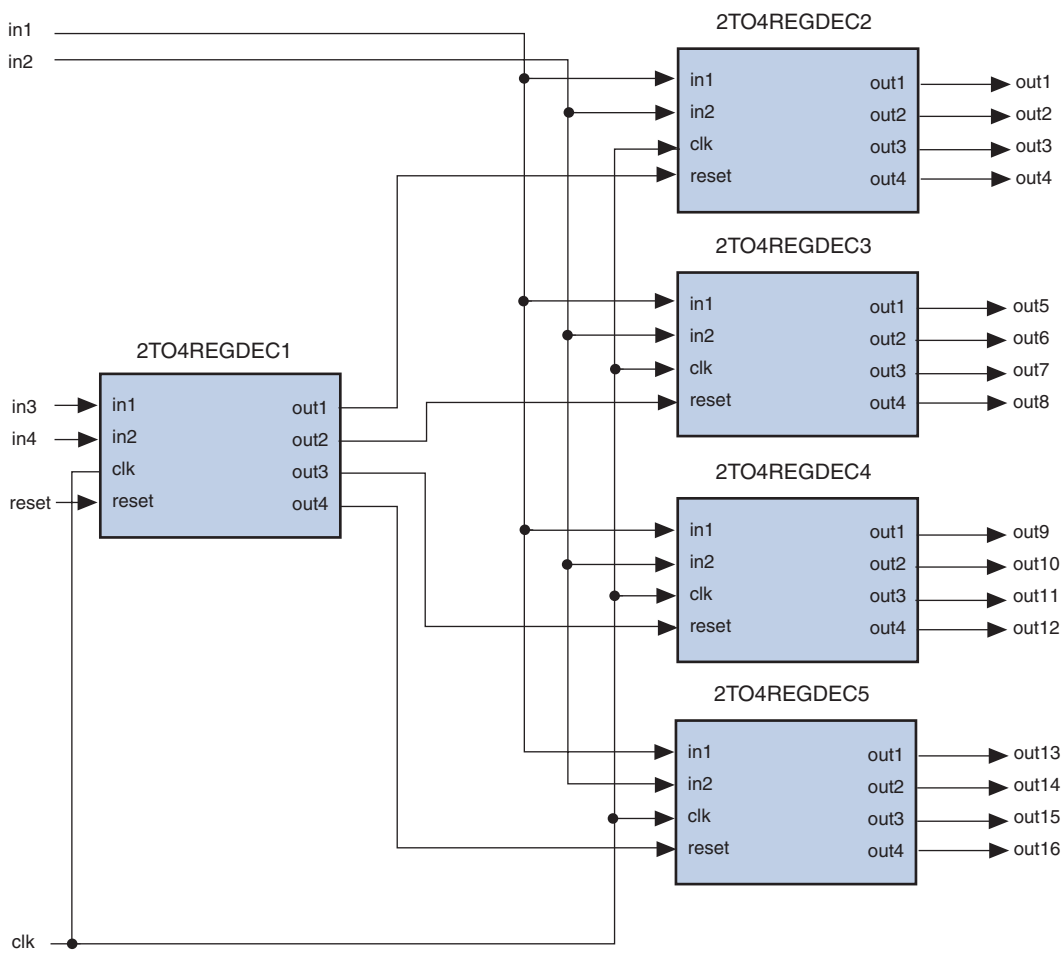


Figure 6–21 shows five 2- to 4-decoders (2TO4REGDEC) that are combined to form a 4- to 16-pipelined decoder. The first decoder (2TO4REGDEC1) will decode the two most significant bits (MSB) (i.e., in3 and in4) of the 4- to 16-decoder. The decoded output from the 2TO4REGDEC1 decoder will only enable one of the rest of the 2- to 4-decoders (i.e., 2TO4REGDEC2, 2TO4REGDEC3, 2TO4REGDEC4, or 2TO4REGDEC5). The inputs in1 and in2 are decoded by the enabled 2- to 4-decoder. Because the time to generate the decoded output increases with the size of the decoder, pipelining the design reduces the time consumed to generate the decoded output, thus improving the maximum frequency. In **Figure 6–21**, the MSBs (i.e., in3 and in4) are decoded in the first clock cycle, while the other bits (i.e., in1, and in2) 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 these techniques may reduce the overall quality of results.

Reducing Synthesis and Synthesis Netlist Optimization Time

You can use Quartus II integrated synthesis to synthesize and optimize HDL designs. You can also use synthesis netlist optimizations to optimize netlists synthesized by third-party EDA software. Using these 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, separately optimize parts of your design hierarchy to reduce analysis and synthesis. Create ATOM netlists for parts of your design you have already synthesized and optimized. The Quartus II Analysis & Synthesis module will not need to reoptimize those netlists, resulting in reduced synthesis and netlist optimization time.



For more information on creating hierarchical designs with multiple netlists, refer to the *Hierarchical Block-Based & Team-Based Design Flows* chapter in Volume 1 of the *Quartus II Handbook*.

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 6–29](#) 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.

Incremental Fitting

Incremental fitting can reduce placement time after an initial compilation because the placer tries to place unchanged nodes in your design in their previous locations. The matching is based on the nodes' logic and connectivity, not just their names. Even if all of the combinational node names have changed, incremental fitting should be able to match the original nodes' functionality and recreate the same placement. Not all nodes need to match, making this mode perfect for Engineering Change Orders (ECOs). Incremental fitting can start an entirely new placement under some conditions:

- More than 500 nodes in the design do not match
- Performance drops by more than 5%
- You significantly change LogicLock regions
- You target a new device
- You delete the design database

Start incremental fitting by choosing **Start > Start Incremental Fitting** (Processing menu).

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 atom netlists. 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 VQM files 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 *Hierarchical Block-Based & Team-Based Design Flows* chapter in Volume 1 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**. 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 6–11 lists the QSF variable name and applicable values for the settings discussed in “Initial Compilation” on page 6–2. 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 6–11. Initial Compilation Settings			
Setting Name	QSF Variable Name	Values	Type
Use Smart Compilation	SPEED_DISK_USAGE_TRADEOFF	SMART, NORMAL	Global
Optimize Timing	OPTIMIZE_TIMING	OFF, “NORMAL COMPIATION”, “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 6–12 lists the QSF variable name and applicable values for the settings discussed in “Resource Utilization Optimization Techniques (LUT-Based Devices)” on page 6–13. 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 6–12. 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_SYNTW_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 6–13 lists the QSF variable name and applicable values for the settings discussed in “I/O Timing Optimization Techniques (LUT-Based Devices)” on page 6–21. 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. 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 6–14 lists the QSF variable name and applicable values for the settings discussed in “ f_{MAX} Timing Optimization Techniques (LUT-Based Devices)” on page 6–27. 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. F_{MAX} Timing Optimization Settings (Part 1 of 2)

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

Table 6–14. F_{MAX} Timing Optimization Settings (Part 2 of 2)

Setting Name	QSF Variable Name	Values	Type
Perform Register Duplication	PHYSICAL_SYNTHESIS_REGISTER_DUPPLICATION	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	DUPLICATE_ATOM	<node name>	Instance

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.

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.

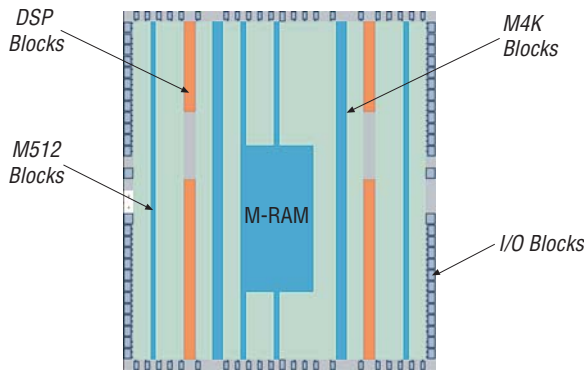
The Altera® Quartus® II software offers many advanced design analysis tools that allow detailed timing analysis of your designs, including a fully integrated Timing Closure Floorplan Editor. With these tools and options, the critical paths in your design can be easily determined and located in the floorplan 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 to perform design analysis.

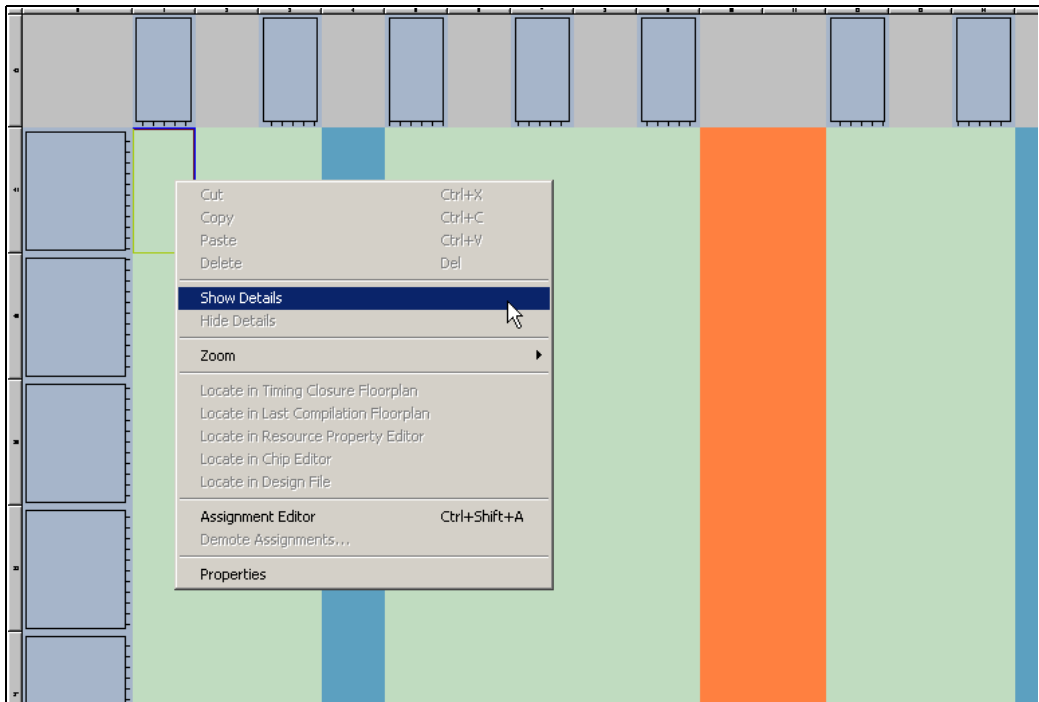
Timing Closure Floorplan Views

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® device.

Figure 7–1. Field View of a Stratix 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.

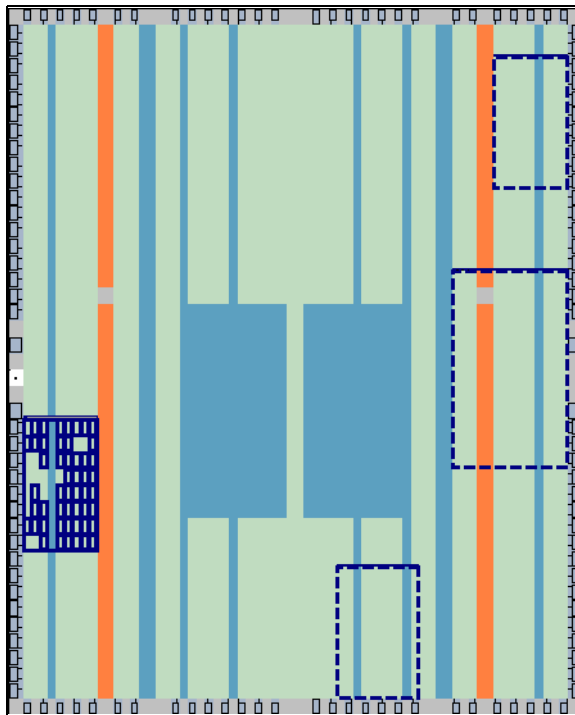
Figure 7–2. Show Details & Hide Details of a LAB in Field View

Viewing Assignments

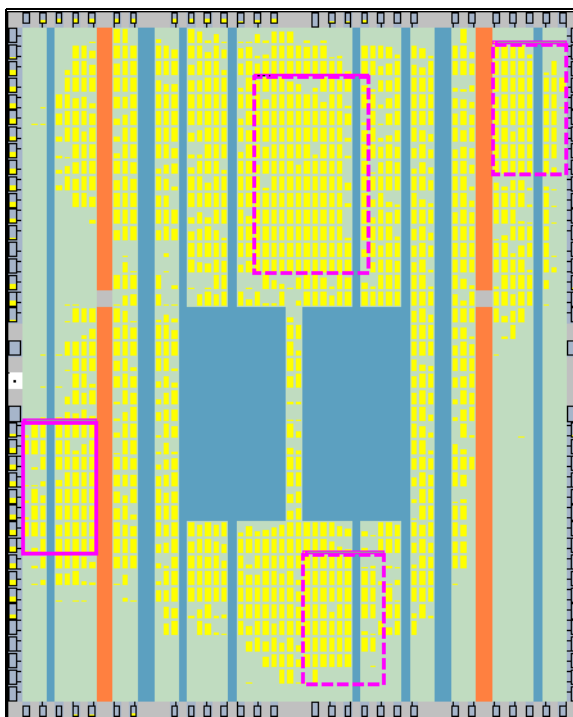
The Timing Closure Floorplan Editor differentiates between user assignments and fitter placements. User assignments are location and LogicLock™ assignments that you make. Fitter placements are the locations where the Quartus II software placed all nodes after the last compilation. You can view both user assignments and fitter placements at the same time.

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 Assignments** 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

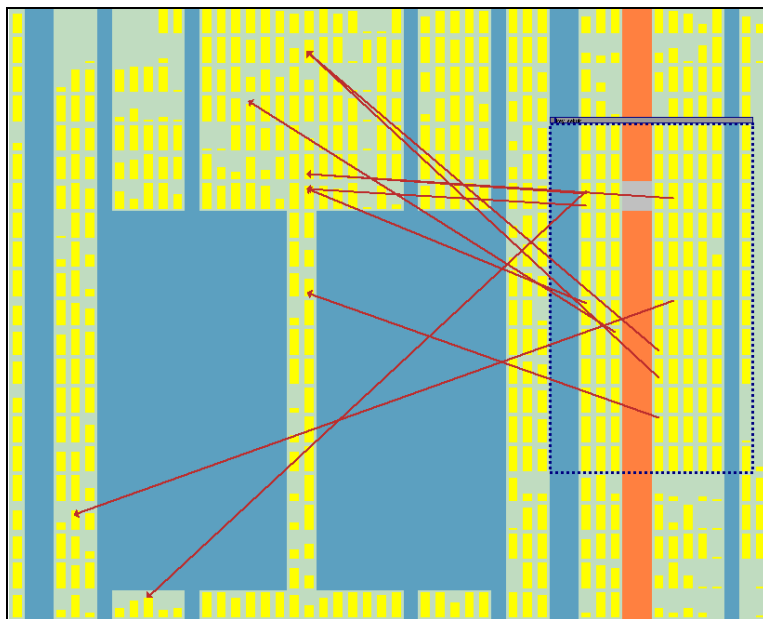
The View Critical Paths feature displays routing paths in the floorplan and ranks their importance, as shown in [Figure 7-5](#). The criticality of a path is determined by either delay or slack. You can view a percentage of critical paths or specify how many paths you wish to see. You can also choose to see paths for all clock domains or a specific clock domain. The following paths can be displayed:

- t_{PD} - The time required for a signal from an input pin to propagate through combinational logic and appear at an external output pin.
- t_{SU} - The length of time for which data that feeds a register via its data or enable input(s) must be present at an input pin before the clock signal that clocks the register is asserted at the clock pin.
- t_{CO} - The maximum time required to obtain a valid output at an output pin that is fed by a register after a clock signal transition on an input pin that clocks the register. This time always represents an external pin-to-pin delay.

- t_H - The minimum length of time for which data that feeds a register through data or enable input(s) must be retained at an input pin after the clock signal that clocks the register is asserted at the clock pin.
- Register-to-Register (f_{MAX}) - The maximum clock frequency that can be achieved without violating internal setup (t_{SU}) and hold (t_H) time requirements.

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 chose **Routing > Critical Paths Settings** (View menu). See [Figure 7-5](#).

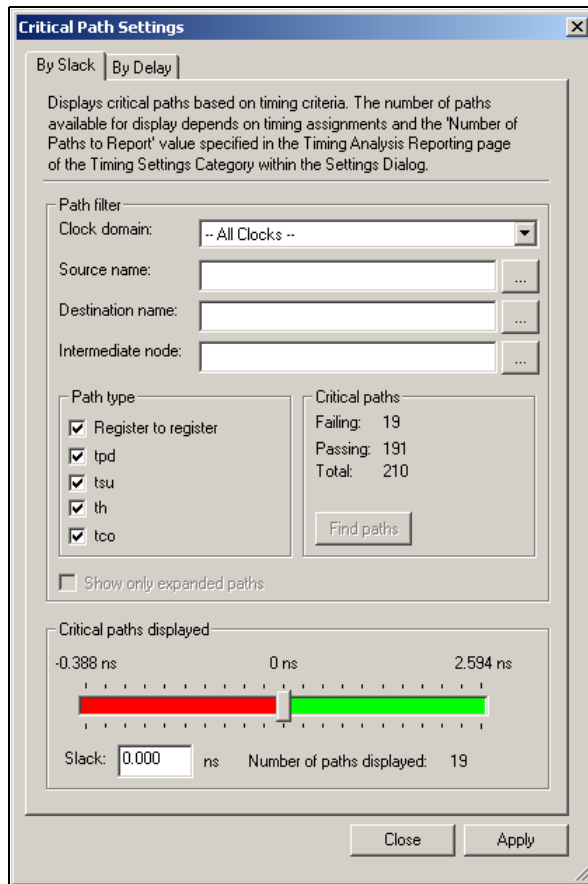
Figure 7-5. Critical Paths



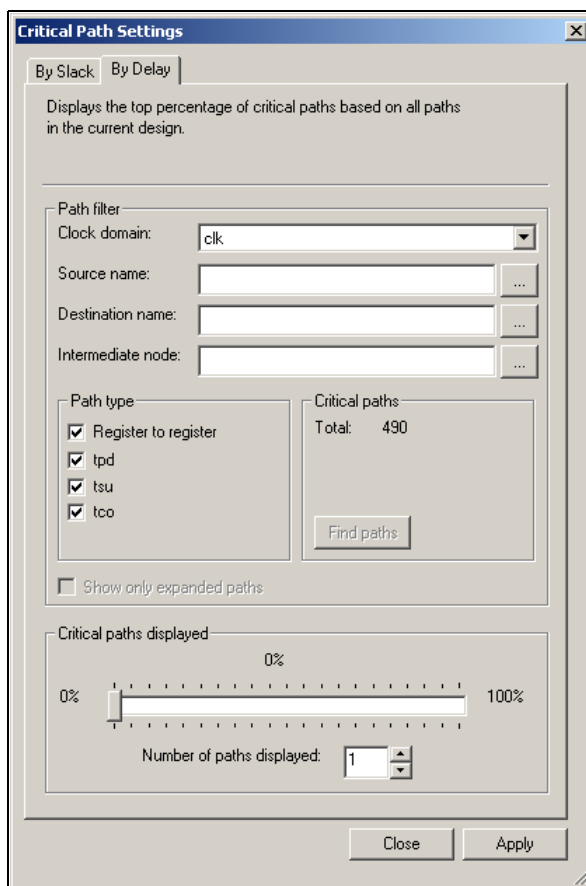
When viewing critical paths by slack, the settings are specified with the **By Slack** tab of the **Critical Path Settings** dialog box shown in [Figure 7-6](#). You determine which path to view and specify the slack threshold beyond which you would like the path displayed in the floorplan. For example, you can view all paths with a slack of -1 ns or worse.



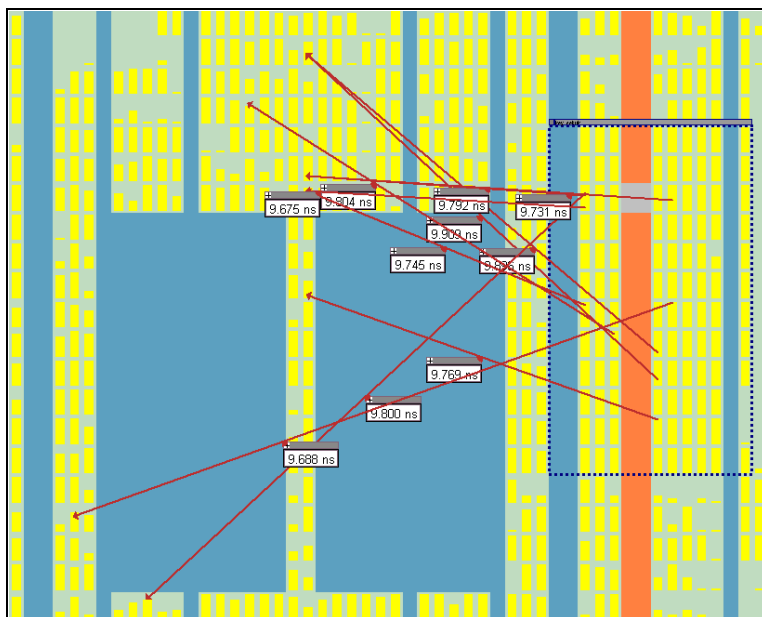
Timing settings must be made for paths to be displayed in the floorplan.

Figure 7–6. Critical Paths Settings, by Slack

When viewing critical paths by delay, the settings are specified with the **By Delay** tab of the **Critical Path Settings** dialog box shown in [Figure 7–7](#). This view displays the critical paths with the longest delay.

Figure 7-7. Critical Paths Settings, by Delay

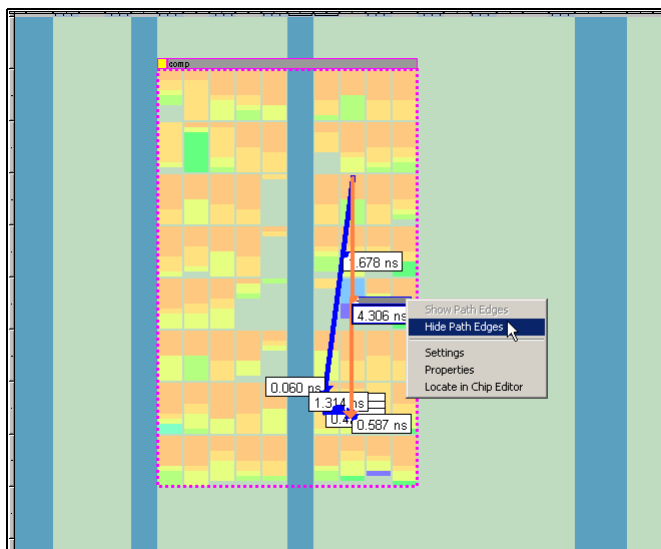
The critical path feature is extremely useful in determining the criticality of nodes based on placement. There are a number of options to view the details of critical path. To see the delay of the critical path, click the **Show Routing Delays** icon or choose **Routing > Show Routing Delays** (View menu). See [Figure 7-8](#).

Figure 7–8. Routing Delays for Critical Paths

The default view shows the path. You can also view all the combinational nodes to see 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 7–9 shows a critical path through combinational nodes. To hide the combinational nodes, select the path, right click, and select **Hide Path Edges**.

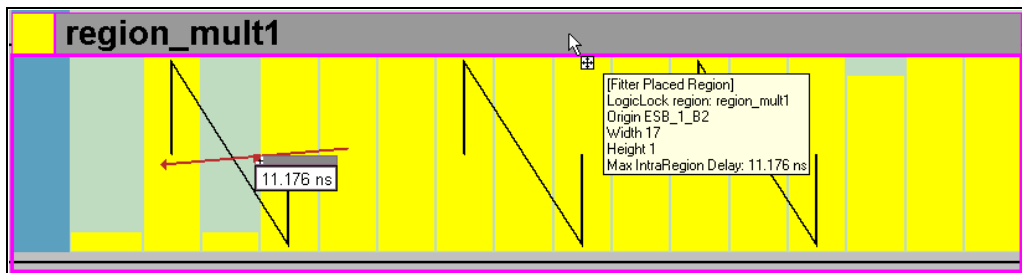


The routing delays must be shown in order to be able to select a path.

Figure 7–9. 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 fitter-placed LogicLock region to see the maximum delay. **Figure 7–10** shows the maximum routing delay of a LogicLock region.

Figure 7–10. Maximum Intra-Region Delay

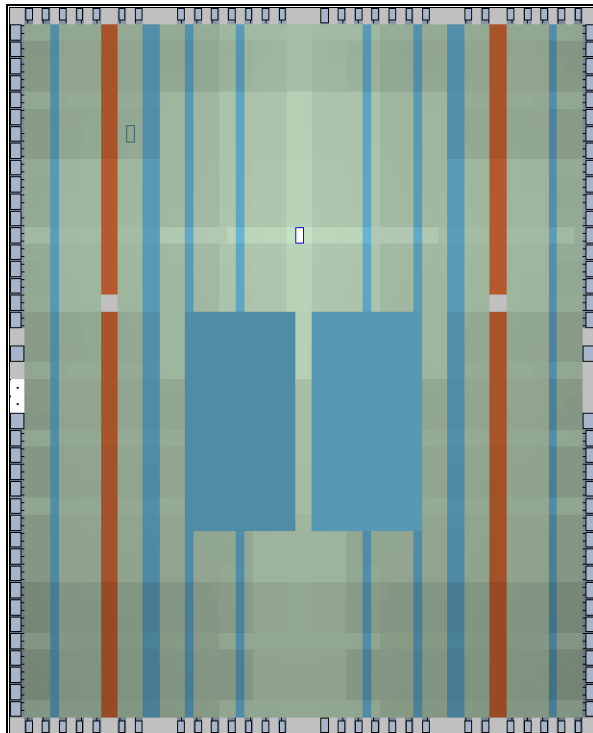


For more information on making path assignments with the **Paths** dialog box, see the *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*.

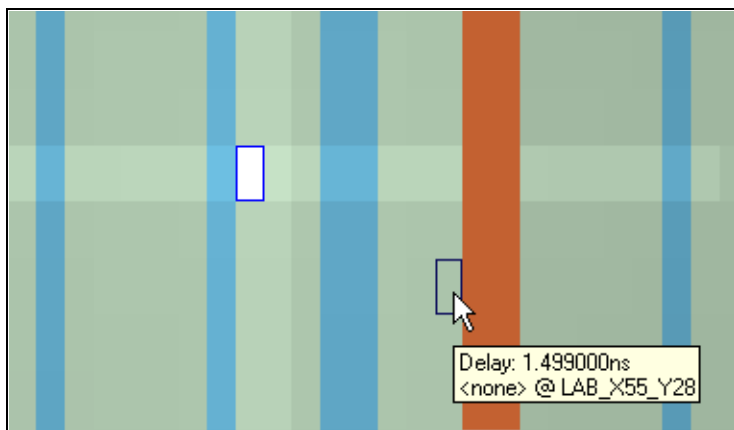
Physical Timing Estimates

In the Timing Closure Floorplan Editor, you can select a resource and see the approximate delay to any other resource on the chip. Once a resource is selected, the delay is represented by the color of potential destination resources. The darker the resource, the longer the delay, as shown in [Figure 7–11](#).

Figure 7–11. Physical Timing Estimates for Large Floorplan



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 shown in [Figure 7–12](#).

Figure 7–12. 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.

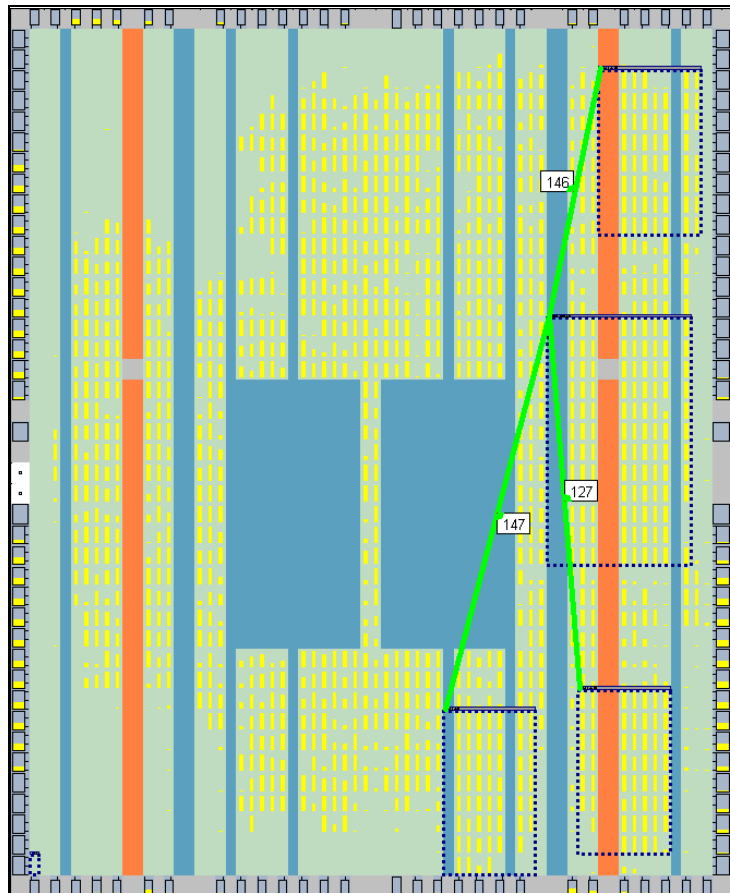
To view the physical timing estimates, click the **Show Physical Timing Estimate** icon or choose **Routing > Show Physical Timing Estimates** (View menu).

The physical timing estimate information can be used 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 assigned 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 7–13 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 7-13. LogicLock Region Connections with Connection Count

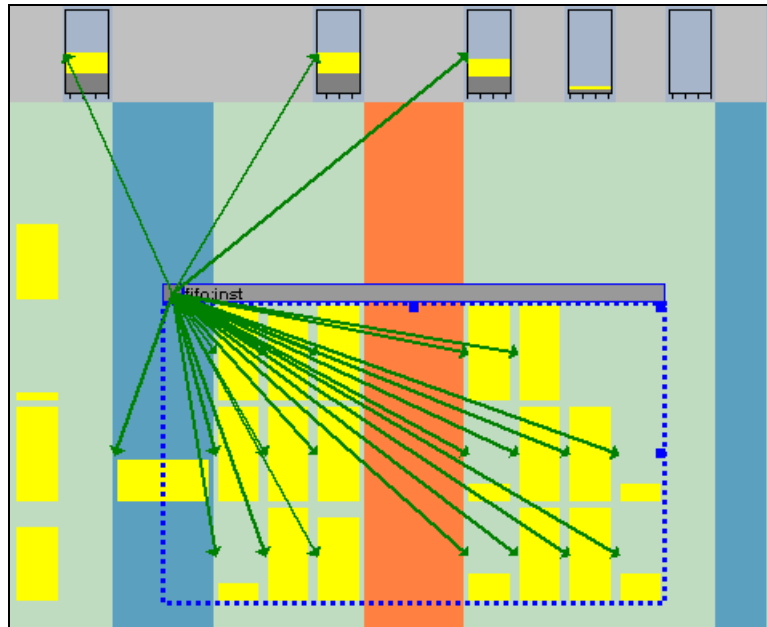
The connection line thickness indicates how many connections exist between regions. To view the number of connections between regions, click the **Show Connection Count** icon or choose **Routing > Show Connection Count** (View menu).

LogicLock region connectivity is applicable only when the user assignments are viewed in the floorplan. When floating LogicLock regions are used, the origin of the user-assigned region is not necessarily the same as the fitter-placed region. This allows you to unlock a region and then lock it down again at a later time. 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 7-14](#) shows the fan-out of a selected LogicLock region.

Figure 7-14. Fan-In or Fan-Out



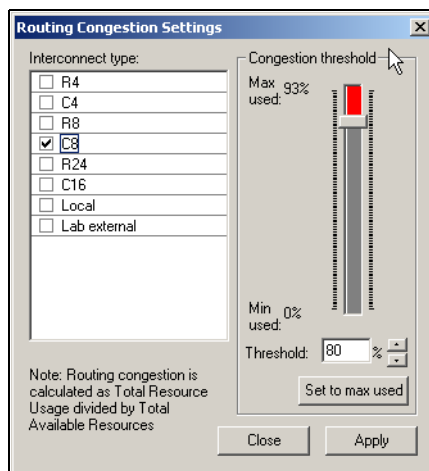
Viewing Routing Congestion

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 7-15 shows the **Routing Congestion Settings** dialog box.

Figure 7-15. Routing Congestion Settings Window



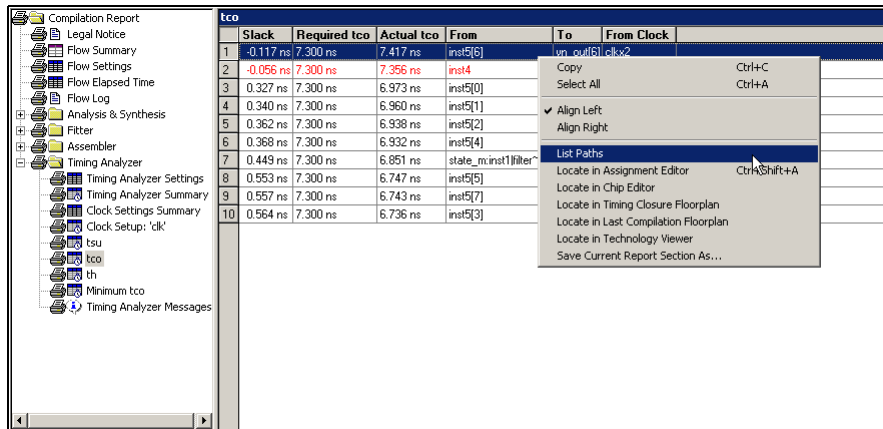
You can choose the routing resource you want to examine and set the congestion threshold. Routing congestion is calculated based on the total resource usage divided by the total available resources.

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 have not met the required timing are reported with a negative slack and are displayed in red as shown in [Figure 7–16](#).

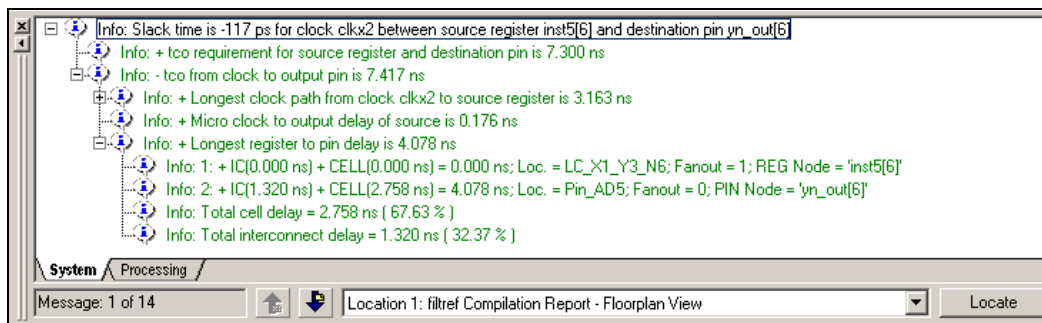
Figure 7–16. I/O Requirements



	Slack	Required tco	Actual tco	From	To	From Clock
1	-0.117 ns	7.300 ns	7.417 ns	inst5[6]	lvn_out[6].clkx2	
2	-0.056 ns	7.300 ns	7.356 ns	inst4		
3	0.327 ns	7.300 ns	6.973 ns	inst5[0]		
4	0.340 ns	7.300 ns	6.960 ns	inst5[1]		
5	0.362 ns	7.300 ns	6.938 ns	inst5[2]		
6	0.368 ns	7.300 ns	6.932 ns	inst5[4]		
7	0.449 ns	7.300 ns	6.851 ns	state_m_inst1filter		
8	0.553 ns	7.300 ns	6.747 ns	inst5[5]		
9	0.557 ns	7.300 ns	6.743 ns	inst5[7]		
10	0.564 ns	7.300 ns	6.736 ns	inst5[3]		

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 7–17](#). This is a good method of determining where along the path the greatest delay is located.

Figure 7-17. 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 7-18 and 7-19. The Timing Closure Floorplan Editor is displayed, highlighting the I/O path. Note that you can set the level of detail in the floorplan in the View menu.

Figure 7-18. Locate Failing Path in Timing Closure Floorplan Editor

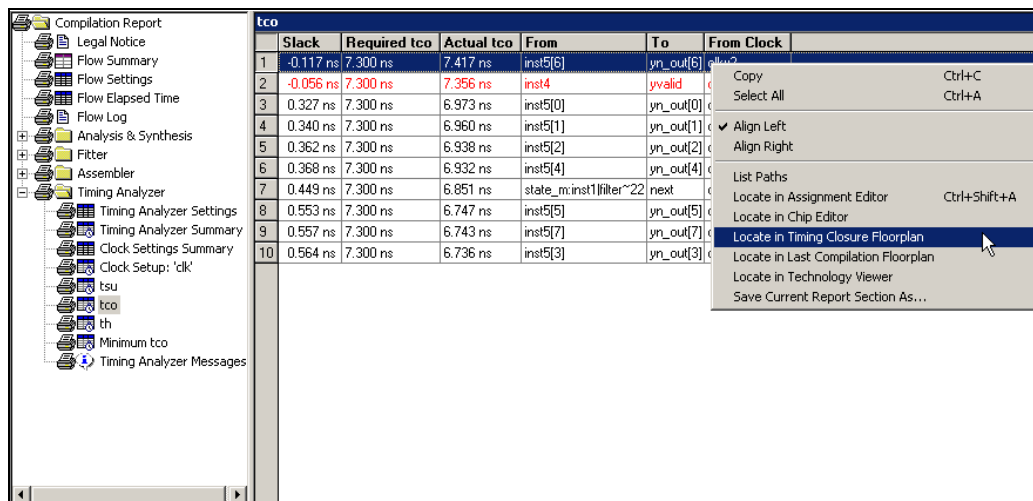
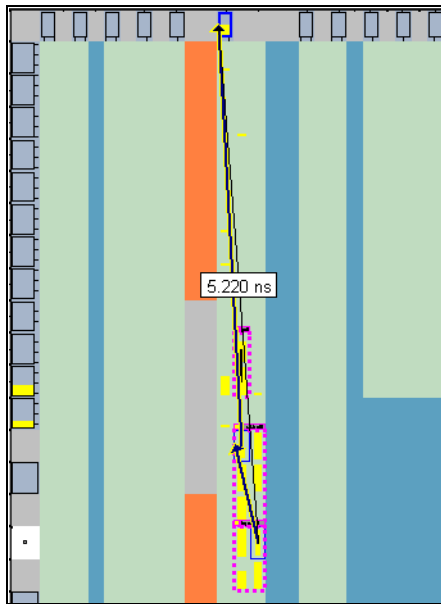
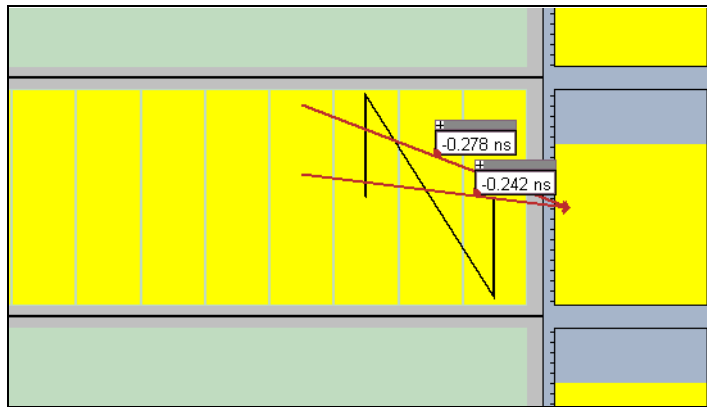


Figure 7-19. Failing Path in Timing Closure Floorplan Editor, Field View

In Figure 7-20 the arrows indicate the critical path (i.e., a register) from the beginning point to the end point (i.e., another register). The times shown are the slack figures for each path. Negative slack indicates paths that failed to meet their timing requirements.

To see all the intermediate nodes (i.e., combinational logic cells) on a path and the delay for each level of logic, right-click the title bar above a path's slack number and choose **Expand** (right button pop-up menu). To view all these paths in the **Timing Closure Floorplan Editor** choose **Routing > Show Critical Paths** (View menu).

Figure 7–20. Critical I/O Paths in the Timing Closure Floorplan



f_{MAX} Timing Analysis Report File

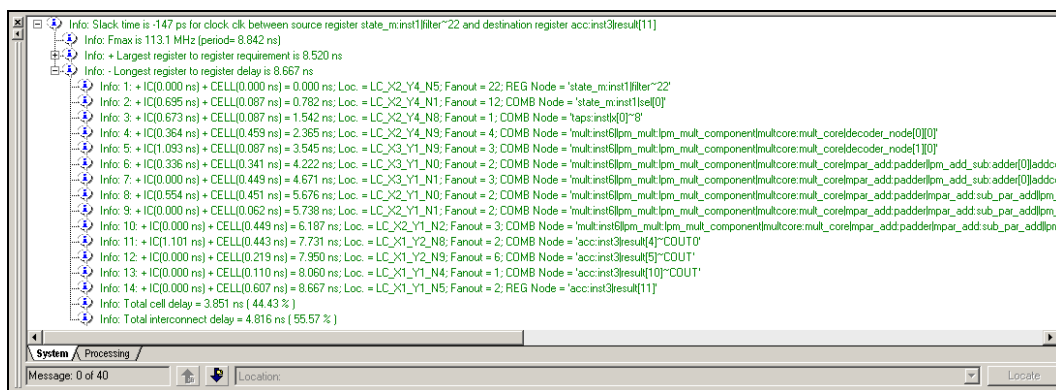
To determine whether your system performance or f_{MAX} timing requirements are met, the Quartus II software generates a timing analysis report that provides detailed timing information on every clock in your design. This report is accessed by opening the **Timing Analyzer** folder in the **Compilation 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 7–21.

Figure 7–21. f_{MAX} Requirements

Clock Setup: 'clk'		From	To	From Clock	To Clock	Requ
Slack	Actual fmax (period)					
1 -0.147 ns	113.10 MHz (period = 8.842 ns)	state_mininst1filter~22	accinst3result[11] all	clk	clk	8.695
2 -0.058 ns	114.25 MHz (period = 8.753 ns)	Copy	Ctrl+C	clk	clk	8.695
3 -0.058 ns	114.25 MHz (period = 8.753 ns)	Select All	Ctrl+A	clk	clk	8.695
4 -0.058 ns	114.25 MHz (period = 8.753 ns)	Align Left		clk	clk	8.695
5 -0.058 ns	114.25 MHz (period = 8.753 ns)	Align Right		clk	clk	8.695
6 -0.058 ns	114.25 MHz (period = 8.753 ns)	List Paths		clk	clk	8.695
7 0.174 ns	117.36 MHz (period = 8.521 ns)	Locate in Assignment Editor	Ctrl+Shift+A	clk	clk	8.695
8 0.191 ns	117.59 MHz (period = 8.504 ns)	Locate in Chip Editor		clk	clk	8.695
9 0.261 ns	118.57 MHz (period = 8.434 ns)	Locate in Timing Closure Floorplan		clk	clk	8.695
10 0.263 ns	118.60 MHz (period = 8.432 ns)	Locate in Last Compilation Floorplan		clk	clk	8.695
11 0.263 ns	118.60 MHz (period = 8.432 ns)	Locate in Technology Viewer		clk	clk	8.695
12 0.263 ns	118.60 MHz (period = 8.432 ns)	Save Current Report Section As...		clk	clk	8.695
13 0.263 ns	118.60 MHz (period = 8.432 ns)	state_mininst1filter~24	accinst3result[7]	clk	clk	8.695
14 0.335 ns	119.62 MHz (period = 8.360 ns)	state_mininst1filter~24	accinst3result[6]	clk	clk	8.695
15 0.337 ns	119.65 MHz (period = 8.358 ns)	taps:inst0[0]~reg0	accinst3result[11]	clk	clk	8.695
16 0.344 ns	119.75 MHz (period = 8.351 ns)	taps:inst0[1]~reg0	accinst3result[11]	clk	clk	8.695
17 0.424 ns	120.90 MHz (period = 8.271 ns)	state_mininst1filter~25	accinst3result[11]	clk	clk	8.695
18 0.424 ns	120.90 MHz (period = 8.271 ns)	taps:inst0[0]~reg0	accinst3result[10]	clk	clk	8.695
19 0.424 ns	120.90 MHz (period = 8.271 ns)	taps:inst0[1]~reg0	accinst3result[9]	clk	clk	8.695
20 0.424 ns	120.90 MHz (period = 8.271 ns)	taps:inst0[0]~reg0	accinst3result[8]	clk	clk	8.695
21 0.424 ns	120.90 MHz (period = 8.271 ns)	taps:inst0[1]~reg0	accinst3result[7]	clk	clk	8.695

To analyze why timing was not met, right-click on a particular path reported in the **System** tab of the **Message** window (Figure 7–22) and select **List Paths** (right button pop-up menu) to determine the location of the greatest delay. You can expand a selection by clicking the "+" icon at the beginning of the line.

Figure 7–22. f_{MAX} Slack Report



Visually analyze f_{MAX} paths by right-clicking on a path in the report and selecting **Locate in Timing Closure Floorplan** to display the Timing Closure Floorplan Editor, which highlights the path. See Figure 7–23. Figure 7–24 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.

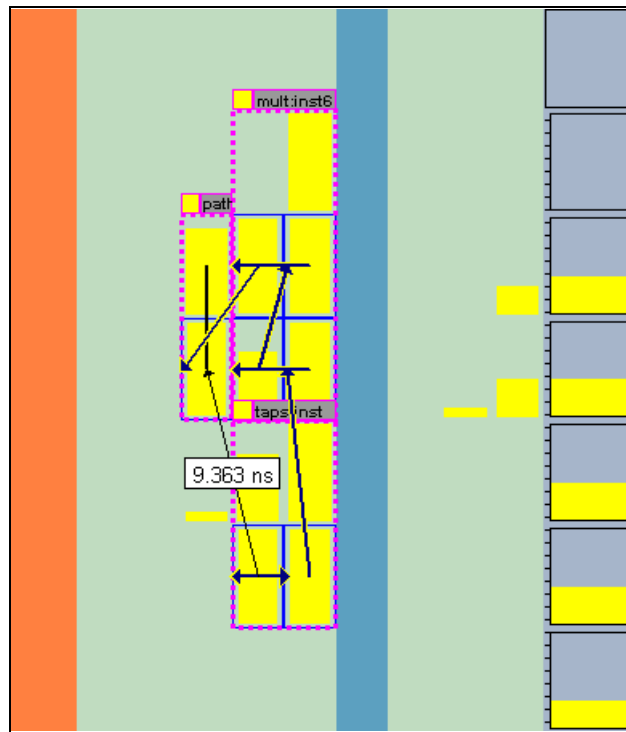
Figure 7–23. Locate Failing Path in Timing Closure Floorplan

The screenshot shows the Timing Analyzer interface with a table of timing paths. A context menu is open over the table, highlighting the 'Locate in Timing Closure Floorplan' option.

Slack	Actual fmax (period)	From	To	From Clock	To Clock	Required Setu...
1 -0.147 ns	113.10 MHz (period = 8.842 ns)	state_minst1filter~22	accinst3result[11] clk	clk	clk	8.695 ns
2 -0.058 ns	114.25 MHz (period = 8.753 ns)	state_minst1filter~22	state_minst1filter~22	clk	clk	8.695 ns
3 -0.058 ns	114.25 MHz (period = 8.753 ns)	state_minst1filter~22	state_minst1filter~22	clk	clk	8.695 ns
4 -0.058 ns	114.25 MHz (period = 8.753 ns)	state_minst1filter~22	state_minst1filter~22	clk	clk	8.695 ns
5 -0.058 ns	114.25 MHz (period = 8.753 ns)	state_minst1filter~22	state_minst1filter~22	clk	clk	8.695 ns
6 -0.058 ns	114.25 MHz (period = 8.753 ns)	state_minst1filter~22	state_minst1filter~22	clk	clk	8.695 ns
7 0.174 ns	117.36 MHz (period = 8.521 ns)	state_minst1filter~24	state_minst1filter~24	clk	clk	8.695 ns
8 0.191 ns	117.59 MHz (period = 8.504 ns)	state_minst1filter~22	state_minst1filter~22	clk	clk	8.695 ns
9 0.261 ns	118.57 MHz (period = 8.434 ns)	state_minst1filter~22	state_minst1filter~22	clk	clk	8.695 ns
10 0.263 ns	118.60 MHz (period = 8.432 ns)	state_minst1filter~24	state_minst1filter~24	clk	clk	8.695 ns
11 0.263 ns	118.60 MHz (period = 8.432 ns)	state_minst1filter~24	state_minst1filter~24	clk	clk	8.695 ns
12 0.263 ns	118.60 MHz (period = 8.432 ns)	state_minst1filter~24	state_minst1filter~24	clk	clk	8.695 ns
13 0.263 ns	118.60 MHz (period = 8.432 ns)	state_minst1filter~24	state_minst1filter~24	clk	clk	8.695 ns
14 0.263 ns	118.60 MHz (period = 8.432 ns)	state_minst1filter~24	state_minst1filter~24	clk	clk	8.695 ns
15 0.335 ns	119.62 MHz (period = 8.360 ns)	taps_instkn[0]*reg0	accinst3result[11] clk	clk	clk	8.695 ns
16 0.337 ns	119.65 MHz (period = 8.358 ns)	taps_instkn[1]*reg0	accinst3result[11] clk	clk	clk	8.695 ns
17 0.344 ns	119.75 MHz (period = 8.351 ns)	state_minst1filter~25	accinst3result[11] clk	clk	clk	8.695 ns
18 0.424 ns	120.90 MHz (period = 8.271 ns)	taps_instkn[0]*reg0	accinst3result[10] clk	clk	clk	8.695 ns
19 0.424 ns	120.90 MHz (period = 8.271 ns)	taps_instkn[0]*reg0	accinst3result[9] clk	clk	clk	8.695 ns
20 0.424 ns	120.90 MHz (period = 8.271 ns)	taps_instkn[0]*reg0	accinst3result[8] clk	clk	clk	8.695 ns
21 0.424 ns	120.90 MHz (period = 8.271 ns)	taps_instkn[0]*reg0	accinst3result[7] clk	clk	clk	8.695 ns
22 0.424 ns	120.90 MHz (period = 8.271 ns)	taps_instkn[0]*reg0	accinst3result[6] clk	clk	clk	8.695 ns
23 0.426 ns	120.93 MHz (period = 8.269 ns)	taps_instkn[1]*reg0	accinst3result[10] clk	clk	clk	8.695 ns
24 0.426 ns	120.93 MHz (period = 8.269 ns)	taps_instkn[1]*reg0	accinst3result[9] clk	clk	clk	8.695 ns
25 0.426 ns	120.93 MHz (period = 8.269 ns)	taps_instkn[1]*reg0	accinst3result[8] clk	clk	clk	8.695 ns
26 0.426 ns	120.93 MHz (period = 8.269 ns)	taps_instkn[1]*reg0	accinst3result[7] clk	clk	clk	8.695 ns

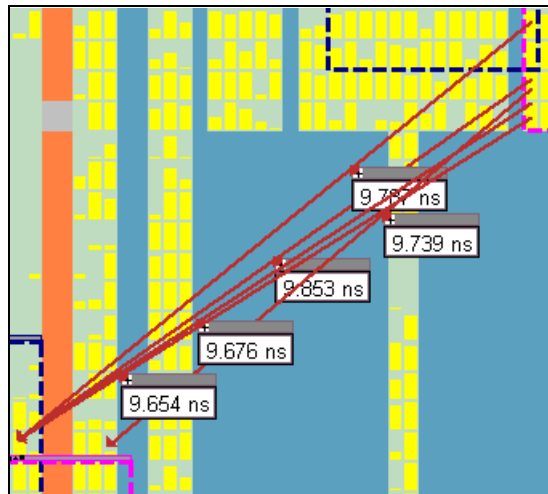
The context menu options are:

- Copy
- Select All
- Align Left
- Align Right
- List Paths
- Locate in Assignment Editor
- Locate in Chip Editor
- Locate in Timing Closure Floorplan
- Locate in Last Compilation Floorplan
- Locate in Technology Viewer
- Save Current Report Section As...

Figure 7–24. 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 7–25 shows critical f_{MAX} paths in the Timing Closure Floorplan Editor.

Figure 7–25. 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.

Introduction

The Quartus® II software offers advanced netlist optimization options, including physical synthesis, to optimize your design further than the optimization performed in the course of the standard Quartus II compilation flow. Device support for these optimizations vary; see the appropriate section for details.

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. These options work with your design's atom netlist, which specifies a design as Altera®-specific primitives. An example of an atom netlist file is an EDIF Input File (.edf) or a Verilog Quartus Mapping (.vqm) file generated by a third-party synthesis tool, or an internal netlist generated within 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. Operating on the output from a third-party synthesis tool, or operating as an intermediate step in the Quartus II standard integrated synthesis, these optimizations make changes to the synthesis netlist. These netlist changes are beneficial in terms of area or speed, depending on your selected optimization technique.

The physical synthesis optimizations take place during the fitter stage of the Quartus II compilation flow. Physical synthesis 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 SignalTap® II or formal verification. Primitive node names are specified during synthesis and are contained in atom netlists from third-party synthesis tools. When netlist optimizations are applied, node names may change as primitives are created and removed. 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*.



Any nodes or entities that have the logic option **Netlist Optimizations** set to **Never allow** are not affected during netlist optimizations (including physical synthesis). This logic option can be applied with the **Assignment Editor** (Assignments menu) if you want to disable all netlist optimizations for parts of your design.

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](#)” on page 8–2 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 be either an EDIF (`.edf`) or VQM (`.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** (Assignments menu)

to assign the **Perform WYSIWYG primitive resynthesis** logic option to a lower-level entity in your design. This option can be used with the Cyclone™ II, MAX® II, Stratix® II, Stratix GX, Stratix, Cyclone, or APEX™ device families.

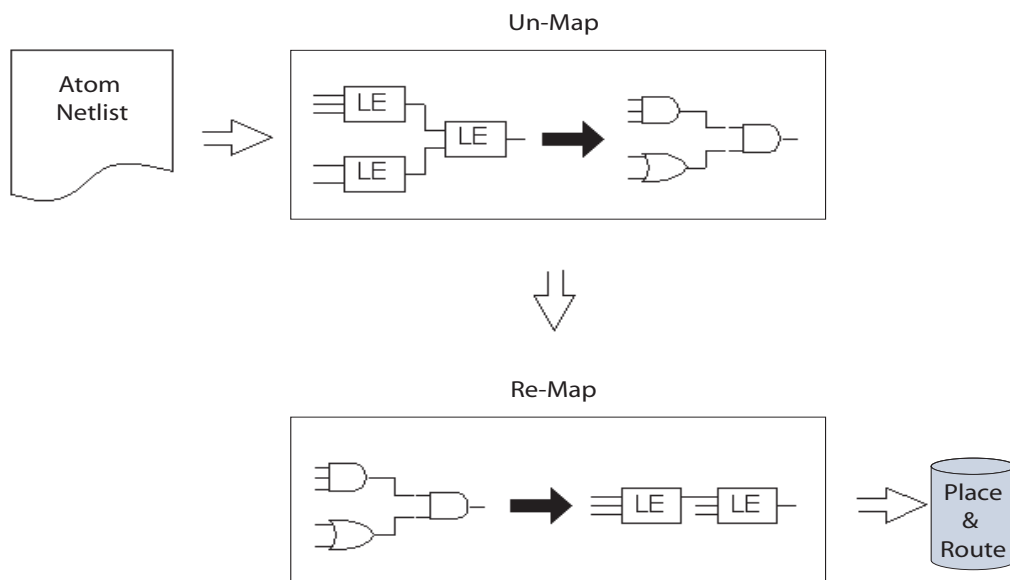
The **Perform WYSIWYG primitive resynthesis** option directs the Quartus II software to un-map the logic elements (LEs) in an atom netlist to logic gates, and then re-map the gates back to Altera-specific primitives. This feature allows the Quartus II software to use different techniques specific to 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 default for most 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



This option is not applicable if you are using Quartus II integrated synthesis. With Quartus II synthesis, you do not need 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 only un-maps and re-maps logic cell (also referred to as LCELL or LE) primitives and regular I/O primitives (which may contain registers). DDR (double data rate) 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

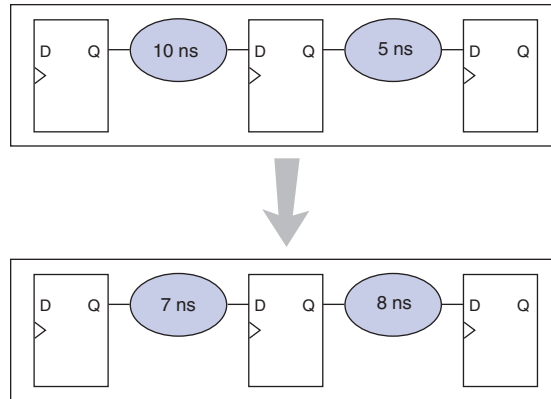
The **Perform gate-level register retiming** option enables movement of registers across combinational logic to balance timing, allowing the Quartus II software to trade off the delay between timing-critical paths and non-critical paths. See [Figure 8–2](#) for an example. It can be used with the Cyclone II, MAX II, Stratix II, Stratix, Stratix GX, Cyclone, and APEX 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.

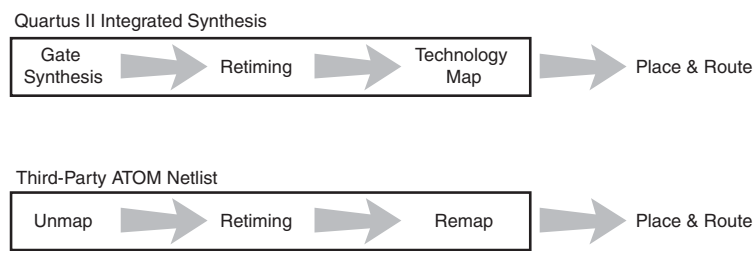
Figure 8–2 shows an example of gate-level register retiming where the 10 ns critical delay is reduced by moving the register relative to the combinational logic.

Figure 8–2. Gate-Level Register Retiming Diagram



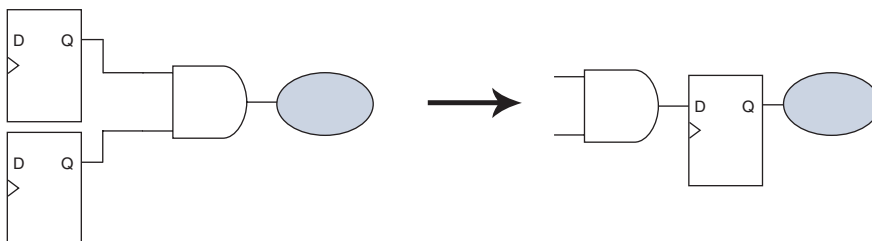
Register retiming makes changes at the gate level. If you are using an atom netlist from a third-party synthesis tool, you must also use the **Perform WYSIWYG primitive resynthesis** option to un-map atom primitives to gates (so that register retiming can be performed) and then to re-map gates to Altera 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 of the benefits 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. This case is shown in [Figure 8–4](#).

Figure 8–4. Combining Registers with Register Retiming

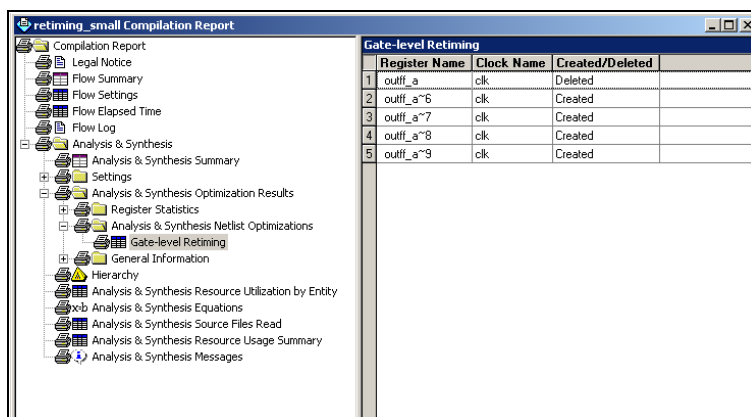
Retiming can only move and combine registers in this type of situation if the following conditions are met:

- All registers have the same clock signal
- All registers have the same clock enable signal
- All registers have asynchronous control signals that are active under the same conditions
- Only one register has an asynchronous load other than VCC or GND

Retiming can always create multiple registers at the input of a combinational block from a register at the output of a combinational block. In this case, the new registers have the same clock and clock enable. The asynchronous control signals and power-up level are derived from previous registers to provide equivalent functionality.

The **Gate-level Retiming** report provides a list of registers that were created and removed during register retiming. 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. Note that the node names for these registers change during the retiming process.

Figure 8–5. Gate-Level Retiming Report



Gate-level Retiming			
	Register Name	Clock Name	Created/Deleted
1	outff_a	clk	Deleted
2	outff_a~6	clk	Created
3	outff_a~7	clk	Created
4	outff_a~8	clk	Created
5	outff_a~9	clk	Created

You can set the **Netlist Optimizations** logic option to **Never Allow** for registers to prevent 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-register-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** on 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 Your Synthesis Netlist Optimization Results

Given the same source code and settings on a given system, the Quartus II software generates the same results on 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. In addition, 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 can 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** page in the **Settings** dialog box (Assignments menu). This option saves your final results as an atom-based netlist in Verilog Quartus Mapping File (.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 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 may lock down the location of all LEs and other 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 elements 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 MAX II, Stratix II, Stratix, Stratix GX, or 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 designers to keep their 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

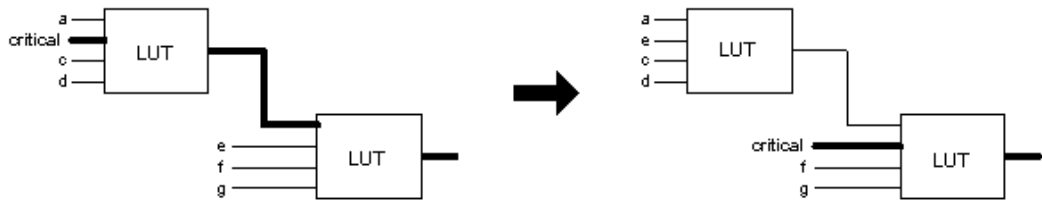
You can control the effect of physical synthesis with the **Physical synthesis effort** option. The default selection is **Normal**. The **Extra** effort setting uses extra compile time to try for extra circuit performance, while the **Fast** effort setting uses less compilation time than **Normal** but may not achieve the same gains.

The **Physical Synthesis** report in the **Fitter Netlist Optimizations** section under **Fitter** in the **Compilation Report** (Processing menu) provides a list of atoms that were modified, created, or deleted during physical synthesis. See the [“Physical Synthesis Report” on page 8–13](#).

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 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 are not swapped.

The Quartus II software does not perform combinational optimization on logic cells that have the following properties:

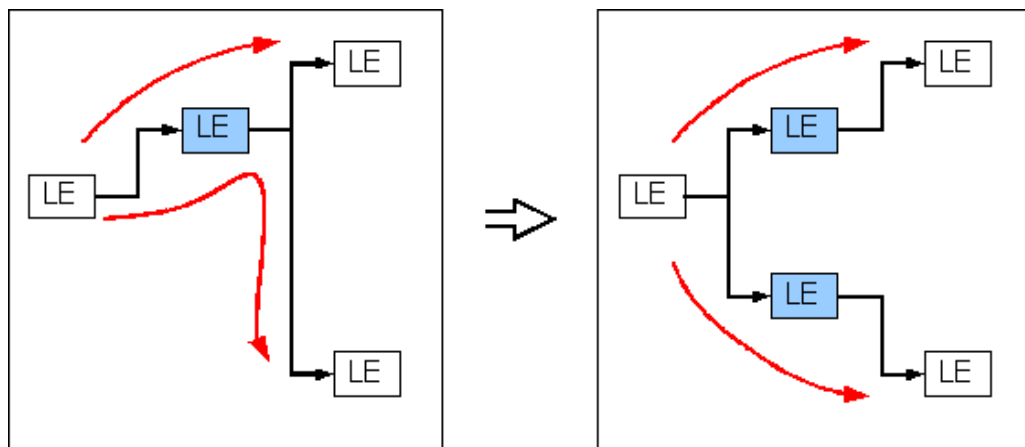
- Are part of a carry/cascade chain
- Drive global signals
- Are constrained to a single logic array block (LAB) location
- Have the **Netlist Optimizations** option set to **Never Allow**

If you want to consider logic cells with any of these conditions for physical synthesis, you can override these rules by setting the **Netlist Optimizations** logic option to **Always Allow** on a given set of nodes.

Physical Synthesis for Registers - Register Duplication

The **Perform register duplication** fitter option on the **Physical synthesis Optimizations** page in the **Fitter Settings** section of the **Settings** dialog box allows the Quartus II fitter to duplicate registers based on fitter placement information. Combinational logic can also be duplicated when this option is enabled. A logic cell that fans out to multiple locations can be duplicated to reduce the delay of one path without degrading the delay of another. The new logic cell may be placed closer to critical logic without affecting the other fan-out paths of the original logic cell.

Figure 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 carry/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 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 compliments the synthesis gate-level option.

The Quartus II software does not perform register retiming on logic cells that have the following properties:

- Are part of a cascade chain
- Contain registers that drive asynchronous control signals on another register
- Contain registers that drive the clock of another register
- Contain registers that drive a register in another clock domain
- Contain registers that are driven by a register in another clock domain
- Contain registers that are constrained to a single LAB location
- Contain registers that are connected to serializer/deserializer (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.

Physical Synthesis Report

All the Physical Synthesis optimizations write results to the **Physical Synthesis** report in the **Fitter Netlist Optimizations** section under **Fitter** in the **Compilation Report** (Processing menu). This report provides a list of atoms that were modified, created, and deleted during physical synthesis. Note that the node names for these atoms change during the physical synthesis process.

Preserving Your Physical Synthesis Results

Given the same source code and settings on a given system, the Quartus II software generates the same results on 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** 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** 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 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 not valid.

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 SignalTap II or formal verification that requires fixed or known node names. In general, 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 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 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) 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, 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 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 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 and Associated Settings (Part 1 of 2)

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 T_{su}/T_{co} with f_{MAX}	ADV_NETLIST_OPT_RETIME_CORE_AND_IO	ON, OFF	Global

Table 8–1. Synthesis Netlist Optimizations and Associated Settings (Part 2 of 2)

Setting Name	QSF Variable Name	Values	Type
Save a node-level netlist into a persistent source file	LOGICLOCK_INCREMENTAL_COMPILE_ASSIGNMENT	ON, OFF	Global
	LOGICLOCK_INCREMENTAL_COMPILE_FILE	<filename>	
Allow Netlist Optimizations	ADV_NETLIST_OPT_ALLOWED	"ALWAYS ALLOW", DEFAULT, "NEVER ALLOW"	Instance

Physical Synthesis Optimizations

Table 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, refer to [“Preserving Your Synthesis Netlist Optimization Results” on page 8–8](#) or [“Preserving Your Physical Synthesis Results” on page 8–14](#).

The following Tcl command back-annotates all registers in your design.

```
logiclock_back_annotate -resource_filter "REGISTER"
```

The `logiclock_back_annotate` command is in the `backannotate` package.

Conclusion

Synthesis Netlist Optimizations and Physical Synthesis Optimizations work in different ways to restructure and optimize your design netlist. Taking advantage of these Quartus II Netlist Optimizations can help improve your quality of results.

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 section describes the Design Space Explorer (DSE), 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.

DSE Concepts

This section provides an explanation of 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 a composition 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 seeds that specify the starting value which 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 value 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 compile 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 defaults to the base compile 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 compile.



The base compilation is the original Quartus II project and is restored after DSE traverses all points in the exploration space.

DSE General Information

You can use DSE in either graphical user interface (GUI) or command-line mode. In either mode, you should run DSE with the Quartus II shell. To run DSE in user interface mode, type `quartus_sh --dse` at a command prompt. To run DSE in command-line mode, type `quartus_sh --dse --nogui <options>` at a command prompt. The example below lists available command-line options.

DSE Command Line Options

Command-line Mode: `quartus_sh --dse -nogui [<options>]`

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-gain *<stop-after-gain value>*
- stop-after-time *<stop-after-time value>*
- ignore-failed-base
- archive
- run-assembler
- slaves *<slave list>*
- use-lsf
- slack-column *<column name>*
- help

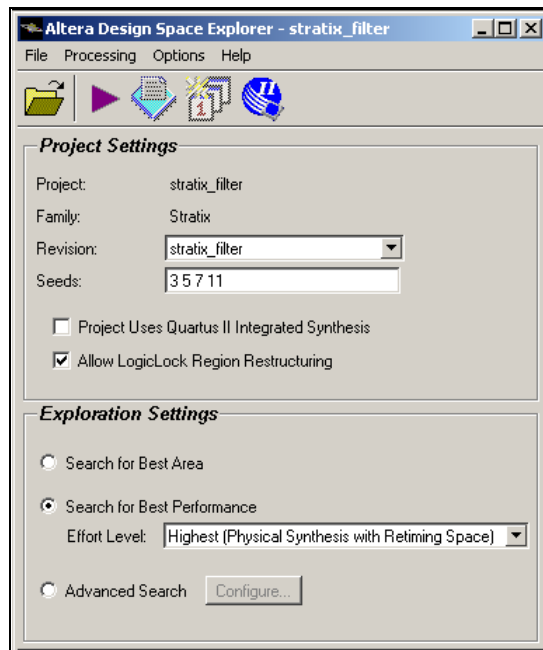
The DSE Tcl/Tk script is in the default Quartus II software installation at `<Quartus II install directory>\bin\tcl_scripts\dse\dse.tcl` on the PC platform and `<Quartus II install directory>/<platform>/tcl_scripts` on the Solaris, HP-UX, and Linux platforms.



For more information, type `quartus_sh --help=dse` at the command prompt.

Figure 9–1 shows the DSE user interface. The main user interface 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 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 launches the Quartus II software for every compilation specified in the Exploration Settings option. 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 compile 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™
- MAX® II

The Advanced Exploration Space supports the following device families, as 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, and FLEX 10K Devices

Seed sweep	Area optimization space
Extra effort space	Signature fitting effort level
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 for the MAX device family is supported only at the command line.

DSE Exploration

DSE compares all exploration space point results with the results of a base compilation. This base compile 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 defaults to the base compile 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 compile.



The base compilation is the original Quartus II project and is restored after DSE traverses all points in the exploration space.

DSE Project Settings

DSE Project Settings

This section includes information about setting up the working environment for DSE, specifying the project and revision, setting the initial seed, and restructuring LogicLock regions.

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. The **Open** icon can also be used to open a project for a design exploration.

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*

The **Seed** field allows you to specify the seed DSE uses in an exploration. The seed value determines the initial placement for your design in a Quartus II compilation.

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 the state of any LogicLock regions in your design.

The section below describes the options available in the **Exploration Settings** section of the DSE user interface.

Use the **Exploration Settings** field to select the type of exploration to perform: **Search for Best Area**, **Search for Best Performance**, or **Advanced Search**.

Use the section “[Exploration Space](#)” on page 9–8 to select the type of exploration to perform: “[Search for Best Area or Performance Options](#)” below, or “[Performing an Advanced Search in Design Space Explorer](#)” on page 9–7.

Search for Best Area or Performance 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 and compilation time increases.

Advanced Search Option

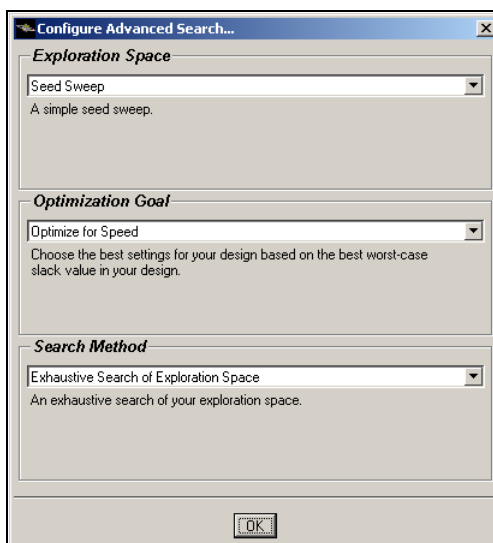
The **Advanced Search** option allows full control over the exploration space, the optimization goal for your design, and the search method used in a design exploration. The section titled “[Performing an Advanced Search in Design Space Explorer](#)” on page 9–7 provides detailed information on how to 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.

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

Allow LogicLock Region Restructuring

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. Also, DSE may remove LogicLock regions that negatively affect the performance of the design.

Exploration Space

The exploration space list controls the exploration type 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 predefined exploration spaces:

- Seed sweep
- Extra effort search
- Physical synthesis search
- Retiming search
- 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 9–16](#).

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, e.g., 2, 3, 4, and 5, or seed ranges, e.g. 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 Search

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.

The **Extra Effort Search for Quartus Integrated Synthesis Projects** exploration space includes all the options in **Extra Effort**, and explores various Quartus II integrated synthesis optimization options. The **Extra**

Effort Search for Quartus Integrated Synthesis Projects exploration space works only for designs that have been synthesized using the Quartus II integrated synthesis.



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

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 Search** 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 **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 Quartus II **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 summaries of exploration spaces, refer to [Table 9–2](#).



For more information about using custom exploration spaces with DSE, see [“Creating Custom Spaces for DSE”](#) on page 9–16.

Table 9–2. Summaries of Exploration Spaces *Note (1)*

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.

Signature Mode

In **Signature** mode, DSE analyzes the f_{MAX} , slack, compile time, and area trade-offs of a single parameter. Running the single parameter over multiple seeds, DSE reports the average of these 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 goals.

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.

The optimization goal is independent of the exploration space. An optimization goal that looks for the best performance, bases its best/worst decisions on the exploration space that produces the highest performance and not one with the smallest logic resource utilization.

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 based on the initial seed specified. This sub-space is then applied to all subsequent 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 run time of the design exploration with the options described in this section.

Continue Exploration Even if Base Compile Fails

DSE continues even if an error occurs during the design compilation. For example, an error occurs in DSE if timing settings are not applied to your design. Turn off this option to make DSE continue with the exploration instead of halting if an error occurs.

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 compilation.

Archive All Compiles

Turn on **Archive All Compiles** 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 compile.

Stop Flow After Gain

Turn on **Stop Flow After Gain** to stop further exploration after a specified percentage gain.

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 the 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

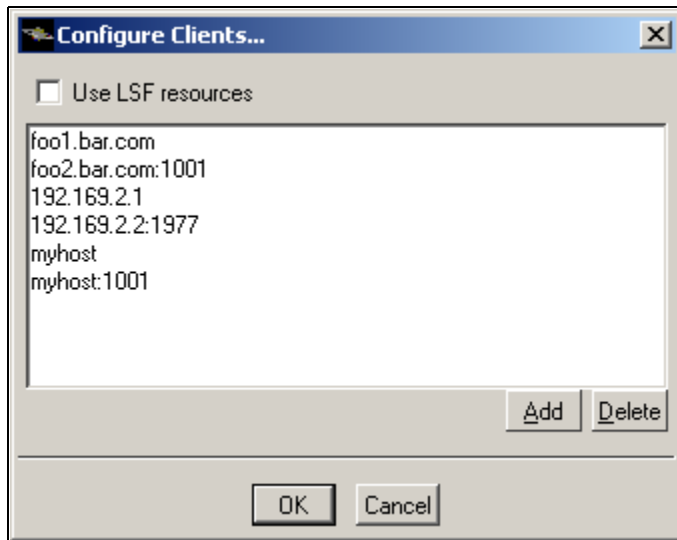
The easiest way to use distributed DSE technology is to submit the compilations to a pre-configured LSF cluster at your local site. For more information on LSF software, see www.platform.com, or contact your system administrator. Turn on **Use LSF resources** to enable this feature.

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. Enter 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 **Enter Clients** dialog box. This dialog box appears after selecting **Exhaustive Search of Exploration Space**. [Figure 9–3](#) shows an example of client entries for a distributed search.

Figure 9–3. 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
- **logiclock**—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 & 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 9–5. Table 9–6 shows the Fitter settings. An example of a POINT tag is shown below:

```
<PARAM name="ADV_NETLIST_OPT_SYNTH_GATE_RETIME"> ON
</PARAM>
```

The point in the example above indicates that the Analysis & Synthesis setting gate-level retiming is turned on for the exploration space point.

Table 9–5. 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 9–5:

(1) Not all Analysis & Synthesis settings are available for all device families.

Table 9–6. 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 9–6. 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 9–6:

(1) Not all Fitter settings are available for all device families.

The custom space example below shows a simple custom exploration space that performs a seed sweep with various Analysis & Synthesis settings and Fitter settings.

Simple Custom Space

```

<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, “Simple Custom Space”, 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, as seen in this example.

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.

The example, “[Custom Space XML Schema](#)”, 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.

Custom Space XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
  <xs:element name="DESIGNSPACE">
    <xs:annotation>
      <xs:documentation>The root element of a design space
description</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="POINT"/>
      </xs:sequence>
      <xs:attribute name="project" type="xs:string" use="optional"/>
      <xs:attribute name="revision" type="xs:string" use="optional"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="POINT">
    <xs:annotation>
      <xs:documentation>A point in the design space</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="PARAM"/>
      </xs:sequence>
      <xs:attribute name="space" type="xs:string" use="required"/>
      <xs:attribute name="enabled" type="xs:boolean" use="optional" default="1"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="PARAM" type="xs:string" nillable="0">
    <xs:annotation>
      <xs:documentation>A single Quartus II software setting</xs:documentation>
    </xs:annotation>
  </xs:element>
</xs: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.

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 version 4.1 supports the LogicLock block-based design flow for the following devices:

- Stratix® II, Stratix, Stratix GX, MAX II®, and Cyclone™
- APEX® and APEX II
- Excalibur™
- Mercury™ (Mercury devices only support 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 Help for instructions on using the LogicLock feature in a sample design.



For more information and guidelines for hierarchical design flow, see the *Hierarchical Block-Based & Team-Based Design Flows* chapter in Volume 1 of the *Quartus II Handbook*.

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 use the LogicLock feature on individual nodes, e.g., to make a wildcard path-based LogicLock assignment on a critical path. This technique is useful if the critical path spans multiple design blocks.



Although LogicLock constraints can improve performance, they can also degrade performance if they are not applied correctly.

Preserving Module Performance

The LogicLock design flow allows you to lock the placement and routing of nodes in a region of a device so that the placement of logic in the region remains constant. The Quartus II software then places the LogicLock region into the top-level design with these constraints.

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 feature, including:

- Creating LogicLock Regions
- Floorplan Editor 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:

- In the **LogicLock Regions** window (Assignments menu)
- Using the **Create New Region** button in the Timing Closure Floorplan
- Using the **Compilation Hierarchy** window
- 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, Stratix GX, Stratix II, MAX II, and Cyclone 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 10–1](#). If you make illegal assignments, you can use the **Repair Branch** command to reset the assignments for the currently selected region and its descendents to legal default values.



For more information on the **Repair Branch** command, see the [“Repair Branch” on page 10–22](#).

Figure 10–1. 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.

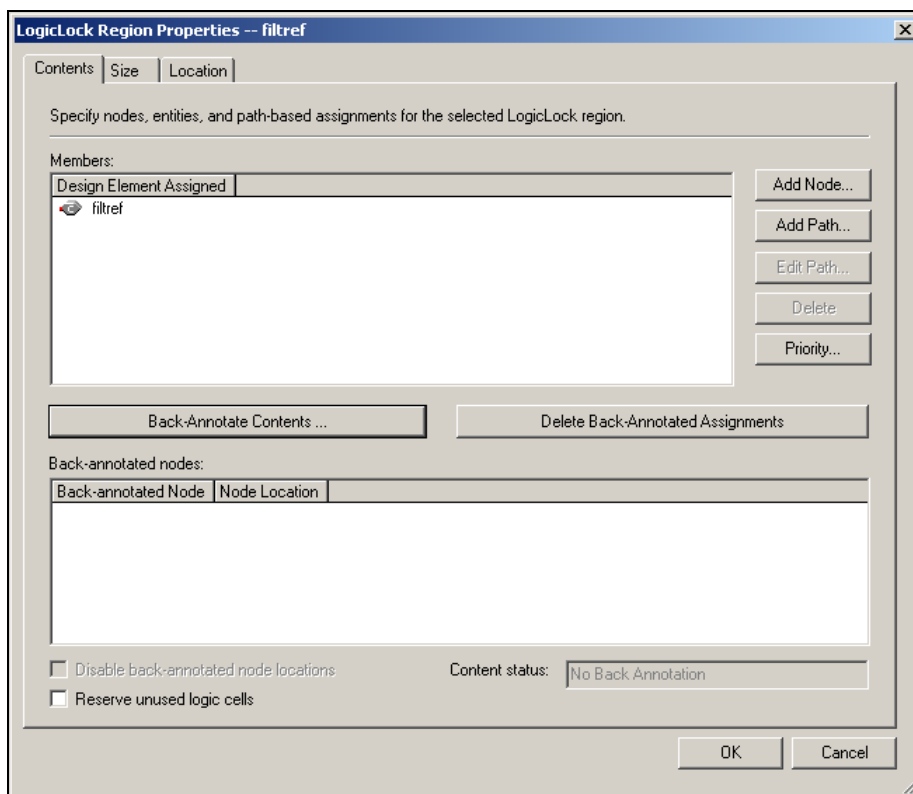
For designs targeting Stratix, Stratix GX, Stratix II, MAX II, and Cyclone devices, the Quartus II software automatically creates a LogicLock region that encompasses the entire device. This default region is labelled `Root_region`, and it is effectively locked and fixed.

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 10–2](#)). 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 10–2. LogicLock Region Properties Dialog Box

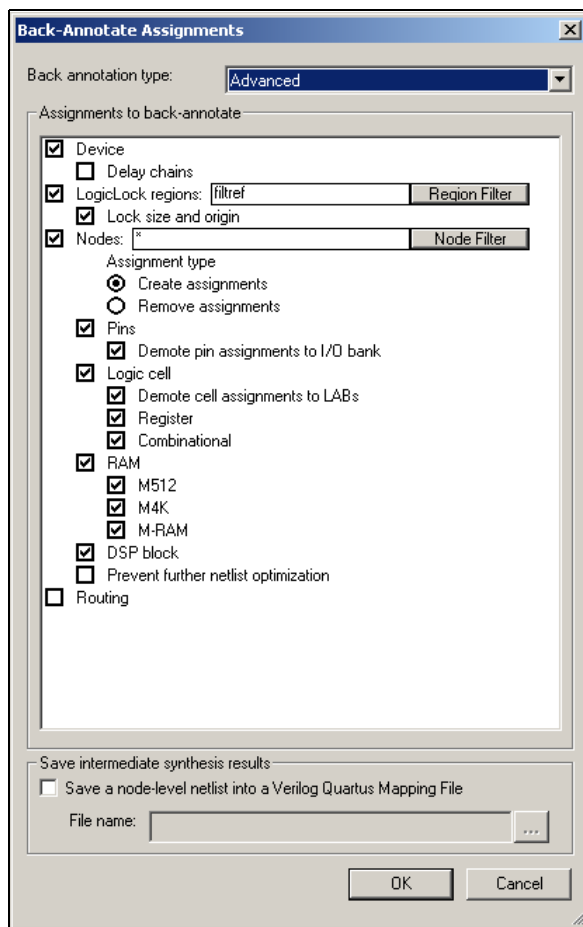


To back-annotate the contents of your LogicLock regions, perform these steps:

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)** (Assignment menu) dialog box (Figure 10–3)
3. Click OK.

Figure 10–3. 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 10–33.

When you back-annotate a region's contents and demote all cell assignments, 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 nodes. If cell assignments are demoted, then the nodes are assigned to LABs rather than directly to logic cells.

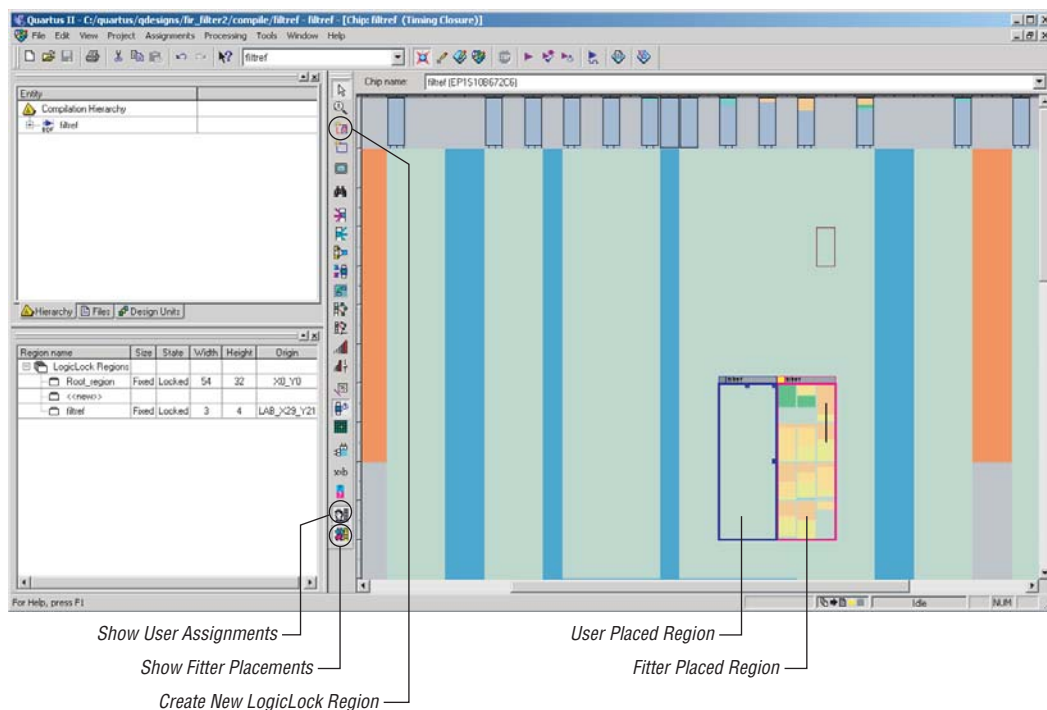
Timing Closure Floorplan Editor

The Timing Closure Floorplan Editor has toolbar buttons with which you can manipulate LogicLock regions, as shown in [Figure 10–4](#). You can use the **Create New 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.

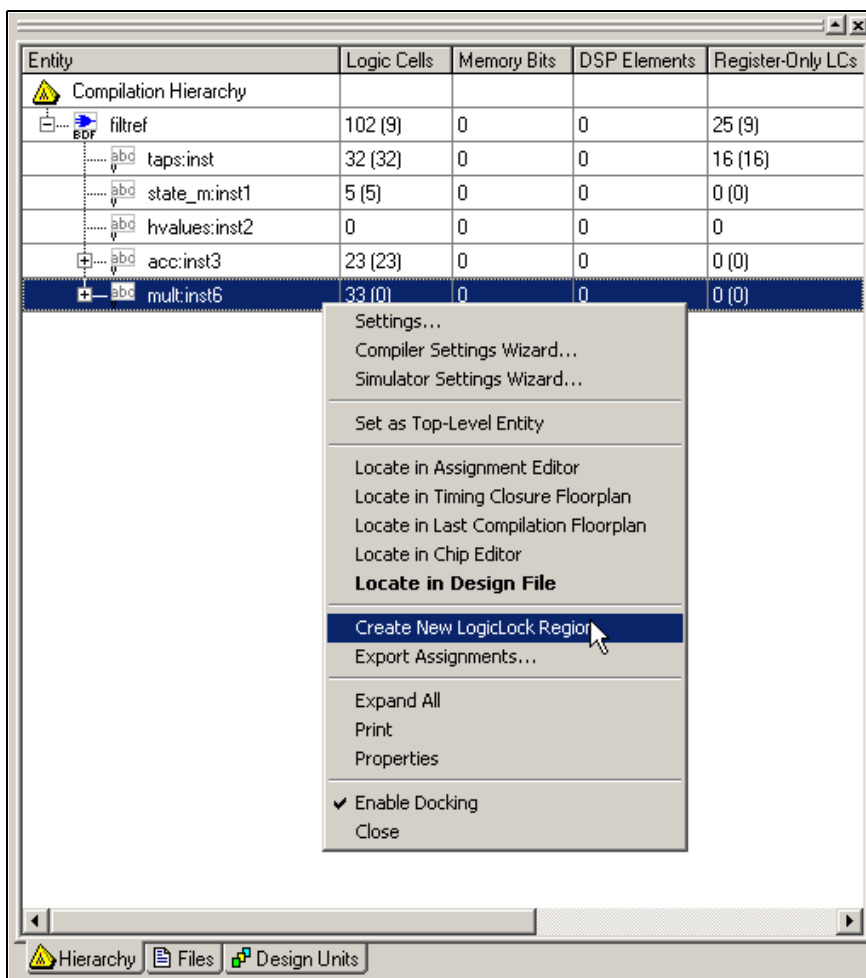
The Timing Closure Floorplan Editor differentiates between user assignments and fitter placements. When the **Show User Assignments** option is turned **on** in the Timing Closure Floorplan, current assignments made to a LogicLock region are visible. When the **Fitter Placement** option is turned **on**, 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 LogicLock border. Fitter-placed regions appear in the Floorplan Editor with a magenta border.

Figure 10–4. Floorplan Editor Toolbar Buttons

Hierarchy Window

After you have performed either a full compilation or Analysis & Elaboration on the design, the Quartus II software displays the hierarchy of the design in the Compilation Hierarchy window. With the hierarchy of the design fully expanded, as shown in [Figure 10–5](#), 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 10–5. Hierarchy Window 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.

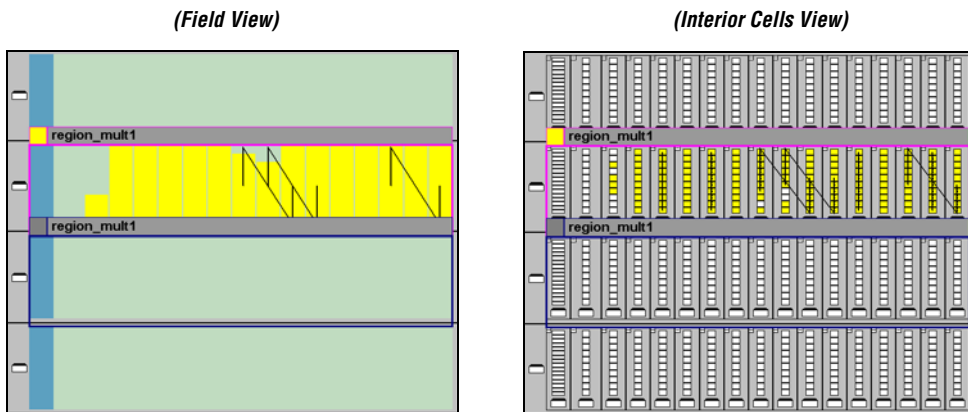


For more information, refer to the *Scripting Support* chapter in Volume 2 of the *Quartus II Handbook*.

Floorplan Editor 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 10–6. The Field View provides an uncluttered view of the device floorplan in which all device resources such as ESBs and MegaLAB™ blocks are outlined. The interior Cells View provides a detailed view of device resources this includes individual Logic Elements within a MegaLAB and device pins.

Figure 10–6. Floorplan Editor—Timing Closure



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

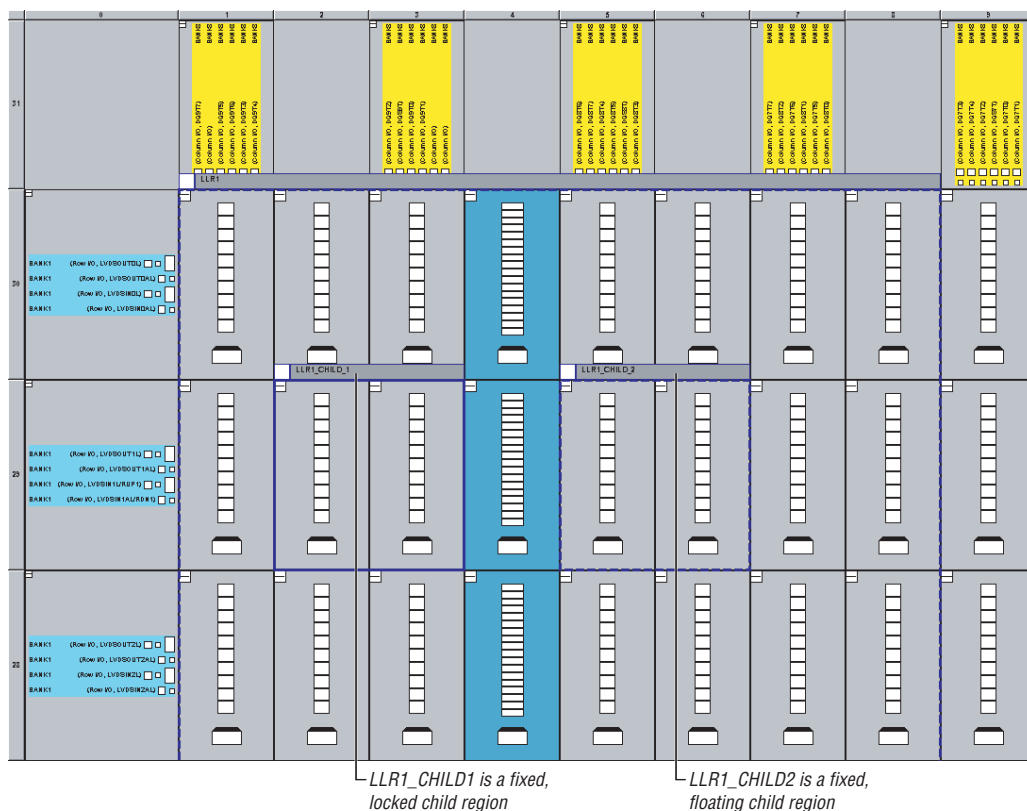
Properties	Values	Behavior
State	Floating (default), Locked	Floating regions allow the Quartus II software to determine the region's location on the device. Locked regions represent user-defined locations of a region and are illustrated with a solid boundary in the graphical floorplans. A locked region 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 you 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.
Enforcement	Hard (default), Soft	Soft 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 regions do not allow contents to be placed outside of the boundaries of the region.
Origin	Any Floorplan Location	The origin defines the top-left corner of the LogicLock region's placement on the floorplan. For Stratix, Stratix II, Stratix GX, MAX 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 only support locked and fixed regions.

The floorplan excerpt in [Figure 10–7](#) shows the LogicLock region properties for a design implemented in a Stratix device.

Figure 10–8. 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.

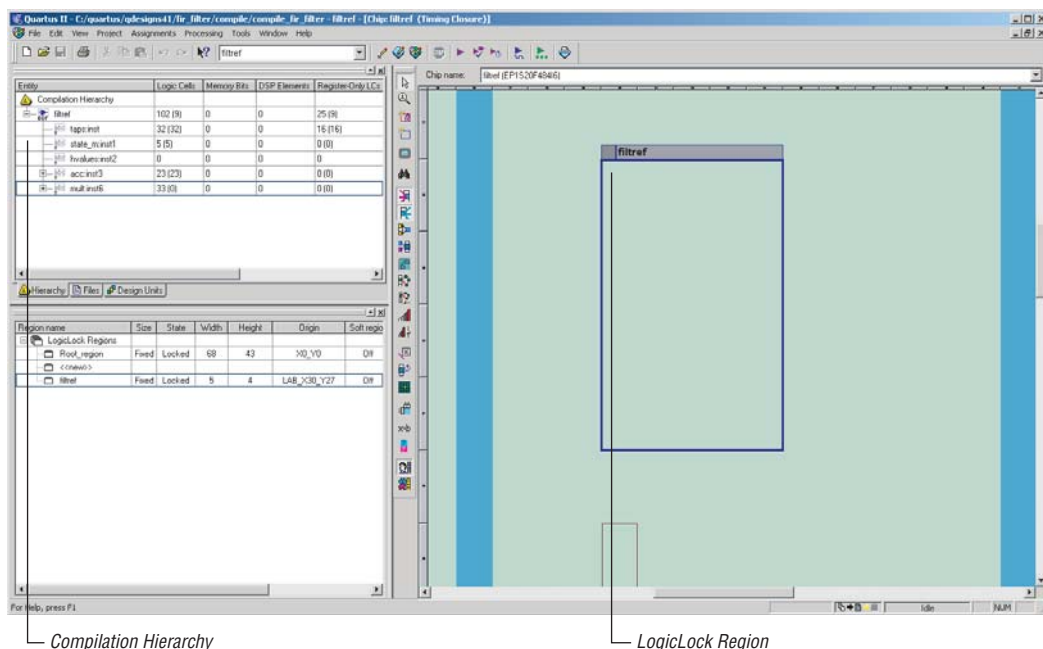
Assigning LogicLock Region Content

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.

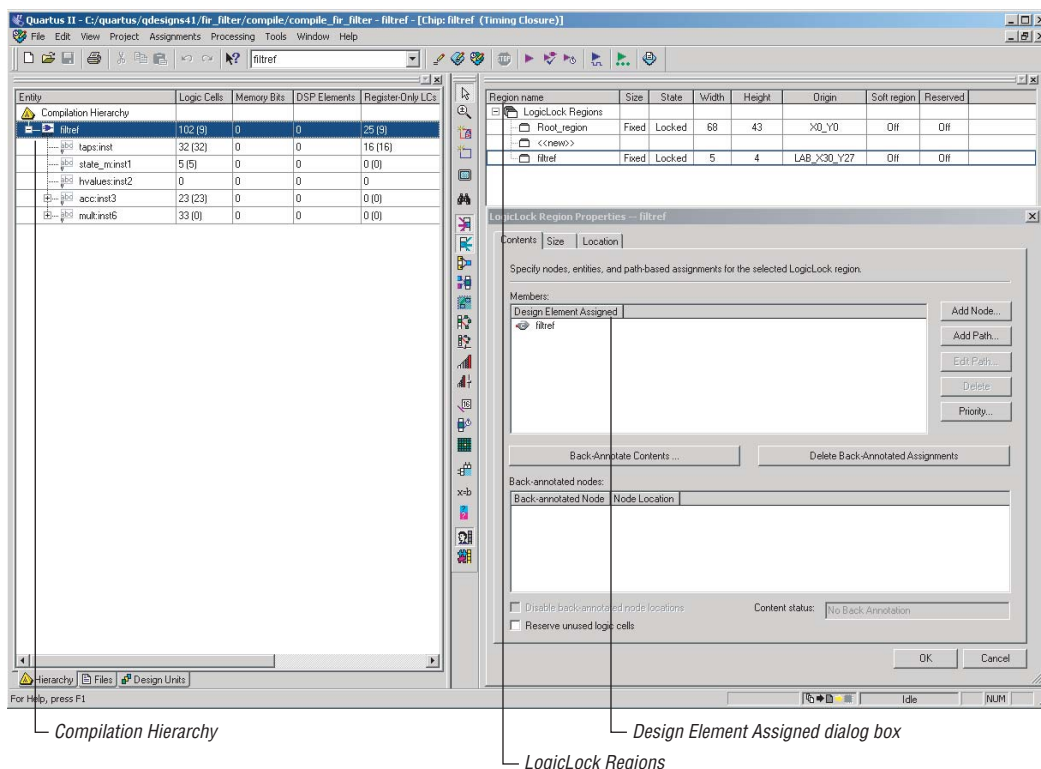
Using Drag & Drop to Place Logic

You can drag selected logic from the Compilation Hierarchy window, Node Finder, or a schematic design file and drop it into the Timing Closure Floorplan or the **LogicLock Regions** dialog box. [Figure 10–9](#) shows logic that has been dragged from the Compilation Hierarchy window dropped into a LogicLock region in the **Timing Closure Floorplan**.

Figure 10–9. Drag & Drop Logic in the Current Assignments Floorplan



[Figure 10–10](#) shows logic that has been dragged from the Compilation Hierarchy window and dropped into the **LogicLock Regions** dialog box. Logic can also be dropped into the **Design Element Assigned** dialog box.

Figure 10–10. Drag & Drop Logic into the LogicLock Regions 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, and Cyclone devices, the locked regions must enclose the I/O pins as resources.

Using the Assignment Editor to Place Logic

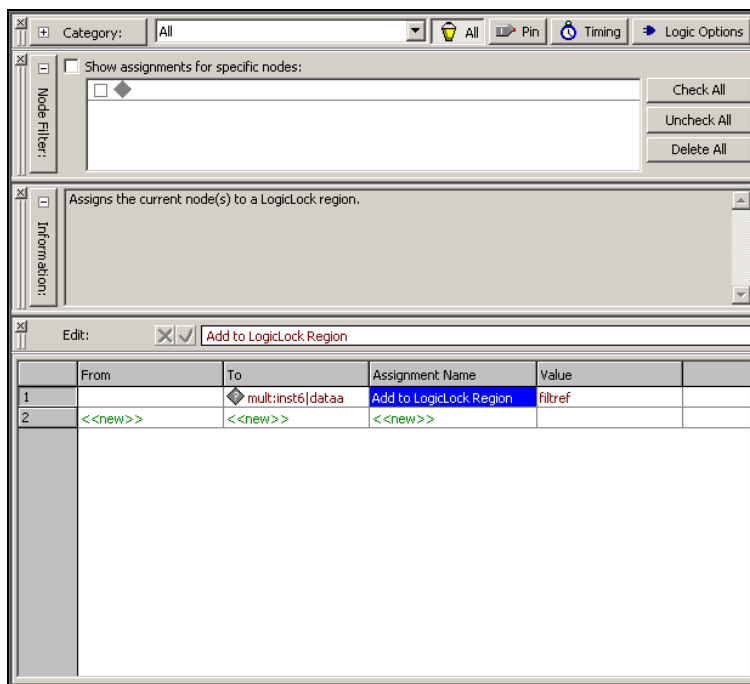
You can also use the Assignment Editor to assign entities and nodes to a LogicLock region (see [Figure 10–11](#)). To assign content to a LogicLock region with the Assignment Editor, perform the following steps:

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 10–11. 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.

Quartus II Block-Based Design Flow

When using the LogicLock design flow, it is recommended that you divide the design into modules. Then, 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®, LeonardoSpectrum™, or FPGA Compiler II software. The software synthesizes each module into an atom netlist, which represents the logic in terms of Altera primitives for the target Altera 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 the [“Constraint Priority” on page 10–30](#) for information on the precedence of the LogicLock region and other constraint settings.

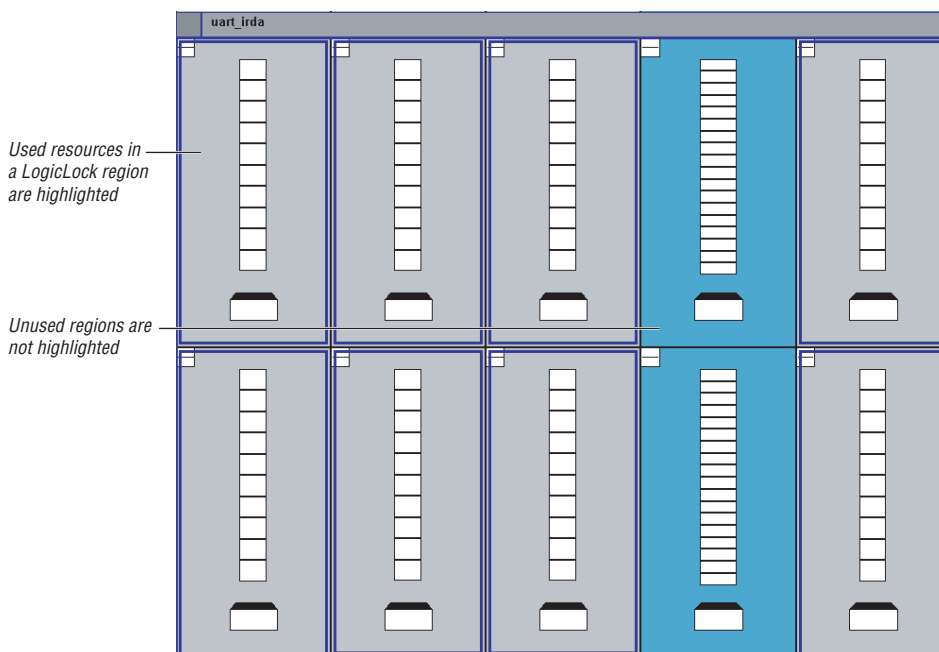
After you have optimized the module so that it meets timing requirements, lock down the placement of nodes in a LogicLock region by back-annotating the contents of the region. To make relative location assignments, you must fix the node names. Fixed node names require an atom netlist so that the assignments 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, e.g., 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 10–12 shows a back-annotated LogicLock region in the **Timing Closure Floorplan**.

Figure 10–12. 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.



It is recommended that you turn on the option **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** option on 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 Compilation Hierarchy 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 and 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 & Elaboration**, or compile the top-level design, so 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 **Analysis & Elaboration**.
3. Expand the design hierarchy on the **Compilation 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.

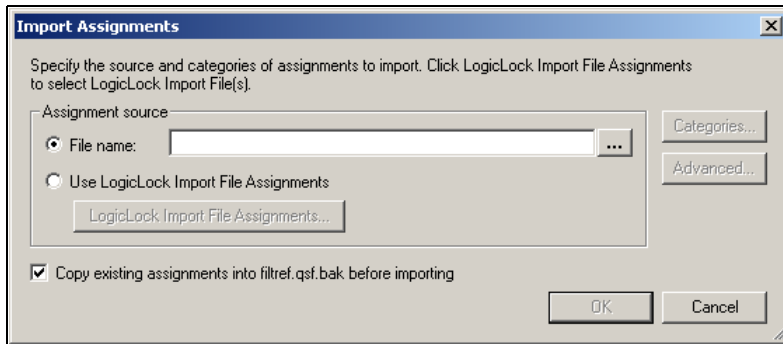
Repeat steps 3 through 5 for all entities that require a specific QSF.

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**.

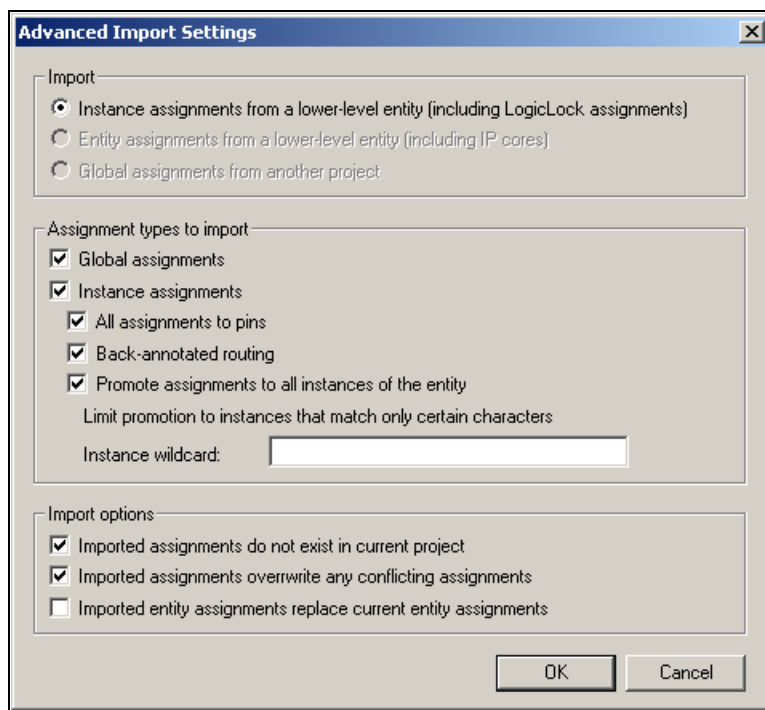
Import the Assignments

To import the assignments, choose **Import Assignments** (Assignments menu). [Figure 10–13](#) shows the **Import Assignment** dialog box.

Figure 10–13. Import Assignments Dialog Box



There are a number of options available in the Advanced Import Assignments dialog box that you can use to control the import of your LogicLock regions, as shown in [Figure 10–14](#).

Figure 10–14. Advanced Import Settings Dialog Box

The Quartus II software converts all imported parent or top-level regions (that do not contain back-annotated routing information) to floating regions to prevent spurious no-fit errors. This allows the Quartus II software to move LogicLock regions to areas on the device with free resources. Child regions are locked or floating relative to their parent region's origin as specified in the modules' 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 so that the pointer is over a LogicLock region name in the Hierarchy window 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 over **Fitter Placed LogicLock Regions** displays the maximum routing delay within the LogicLock region. You must first turn on the **Show Critical Paths** (see [“Show Critical Paths” on page 10–24](#)) command before the delay information becomes available.

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** options exclude any arbitrary 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.



The **Prevent Assignment to LogicLock Regions** option for a given entity or node is found 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.

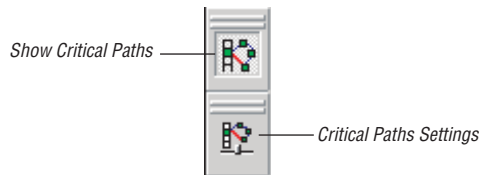
Rubber Banding

When the **Rubber Banding** option is turned on, the Quartus II software shows 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 either one or more of the following paths: pin-to-pin, pin-to-register, register-to-pin, or register-to-register, as shown in [Figure 10–15](#).

Figure 10–15. 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

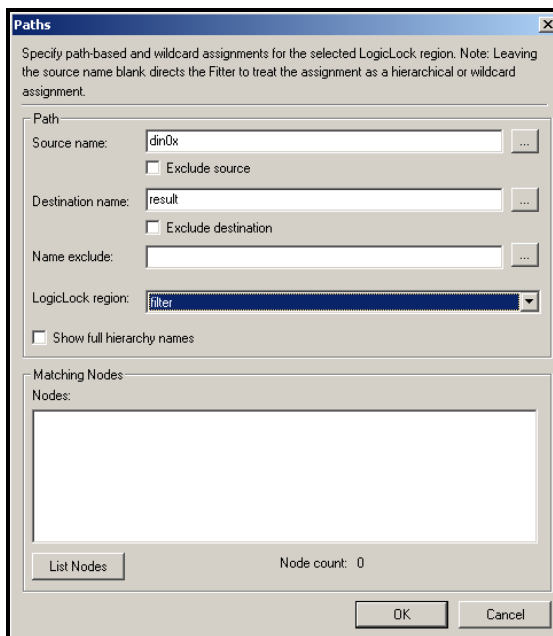
The Compilation Report contains an **Analysis & Synthesis Resource Utilization by Entity** section, which reports accurate resource usage statistics, including entity-level information. This feature is useful when manually creating LogicLock regions.

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 10–16](#) shows the **Path-Based Assignment** dialog box.



Both "*" and "?" wildcard characters are allowed for both the source and destination nodes. When creating path-based assignments you can have the option of excluding certain nodes with the Name exclude field in the **Path** dialog box.

Figure 10–16. Path-Based Assignment Dialog Box

The **Path-Based Assignment** dialog box is launched from the **Contents Tab** of the **LogicLock Regions** 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 versus Soft LogicLock Regions

Normally all nodes assigned to a particular LogicLock region always resides 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,” 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 parent region. If a non-soft parent does not exist (in a design targeting a Stratix, Stratix GX, Stratix II, MAX II, or Cyclone device), the region floats within the `Root_region`, i.e., 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.

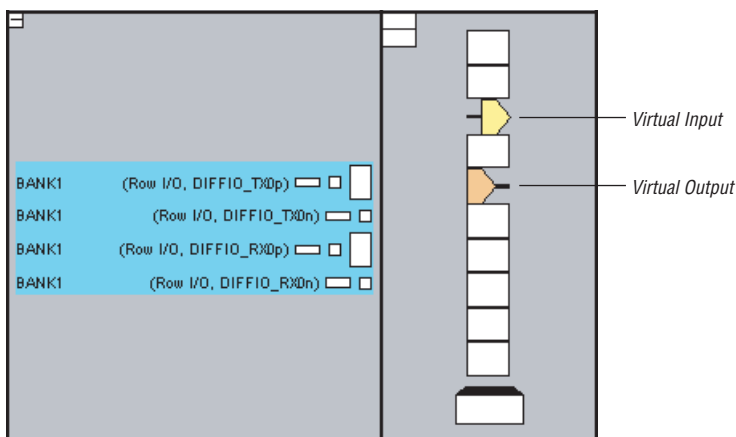
Soft LogicLock regions cannot be back-annotated because the Quartus II software may have placed nodes outside of the LogicLock region resulting in undefinable location assignments relative to the region’s origin and size.

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 10–17](#) shows the virtual input and output pins in the **Floorplan Editor**.

Figure 10–17. 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.

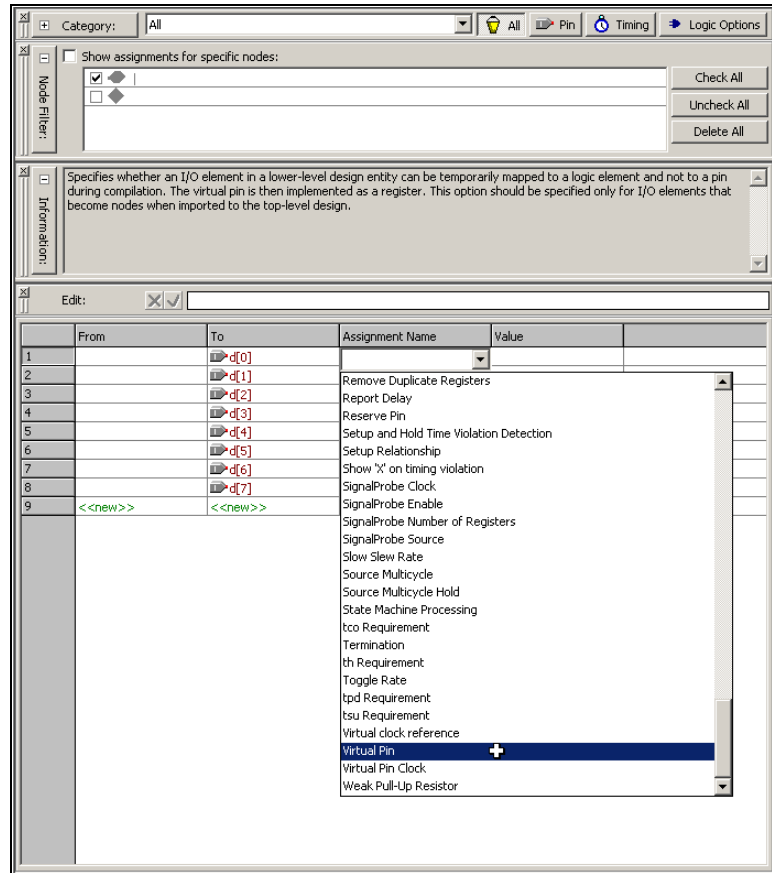
In the top-level design, these virtual pins are connected to an internal node in another module. Making assignments to virtual pins allow you to place them within the same location or region on the device as the corresponding internal node would exist in the top-level module. This feature also has the 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 to Virtual Pin Clock option on for each virtual pin.



Virtual pin and virtual pin clock assignments are made through the **Assignment Editor**. [Figure 10–18](#) shows assigning virtual pins using the **Assignment Editor**.

Figure 10–18. Using the Assignment Editor to Assign Virtual Pin



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:

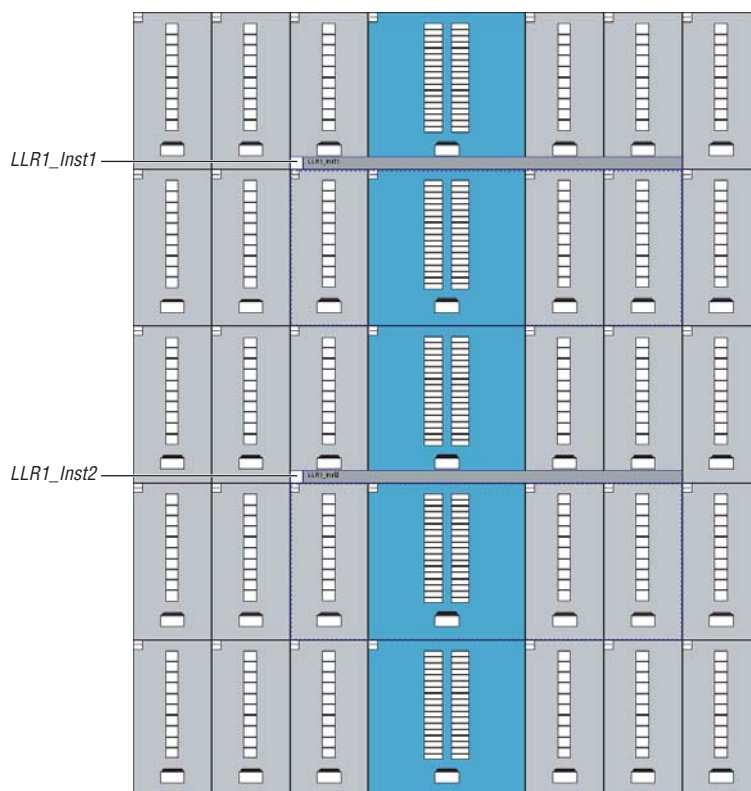
- LogicLock regions with pin assignments must be placed on the periphery of the device, adjacent to the pins. (For Stratix, Stratix GX, Stratix II, MAX II, and Cyclone devices, you must also include the I/O block.)
- Floating LogicLock regions cannot overlap.
- It is recommended that you not create fixed 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 10–19 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 in the devices that have the same resources.

Figure 10–19. Floorplan of Two Instances of a LogicLock Region



Notes for Figure 10–19:

- (1) The back-annotated regions LLR1_Inst1 and LLR1_Inst2 have the same resources.

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.

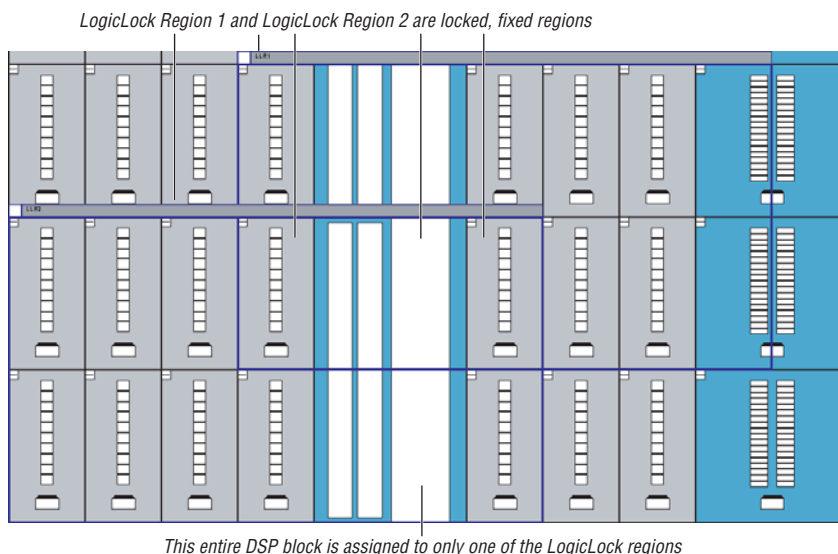


Pin assignments to LogicLock regions honor only fixed and locked regions. Pins assigned to floating regions do not influence the region's placement.

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 10–20 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 10–20. Overlapping LogicLock Regions in a Stratix DSP Block



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 10–3 on page 10–5](#)).



If you are not using an atom netlist, you must turn On the **Save a node-level netlist into a Verilog Quartus Mapping File** option On in the **Back-Annotate Assignments** 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 between source and destination logic. The VQM is then be 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

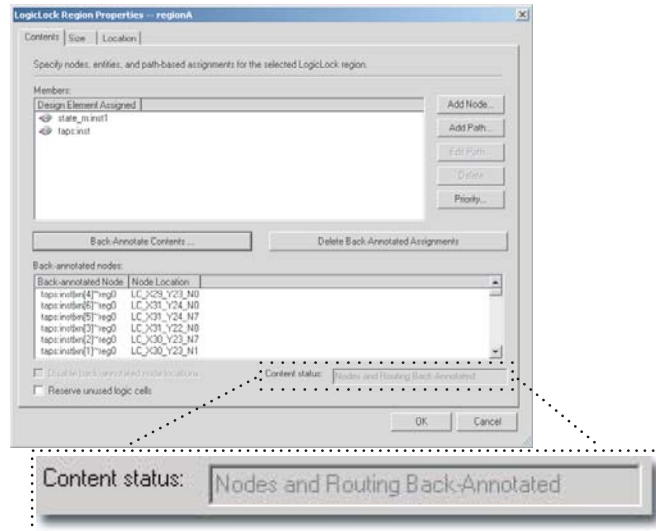
You can export the LogicLock region routing information by turning On the **Export Back-annotated routing** option On in the **Export Assignments** dialog box (Assignments menu). This generates a QSF and a RCF in the specified directory. The QSF file contains all LogicLock region properties as specified in the current design. The RCF contains all the necessary routing information for the exported LogicLock regions.

This RCF only works with the atom netlist for the entity being exported.

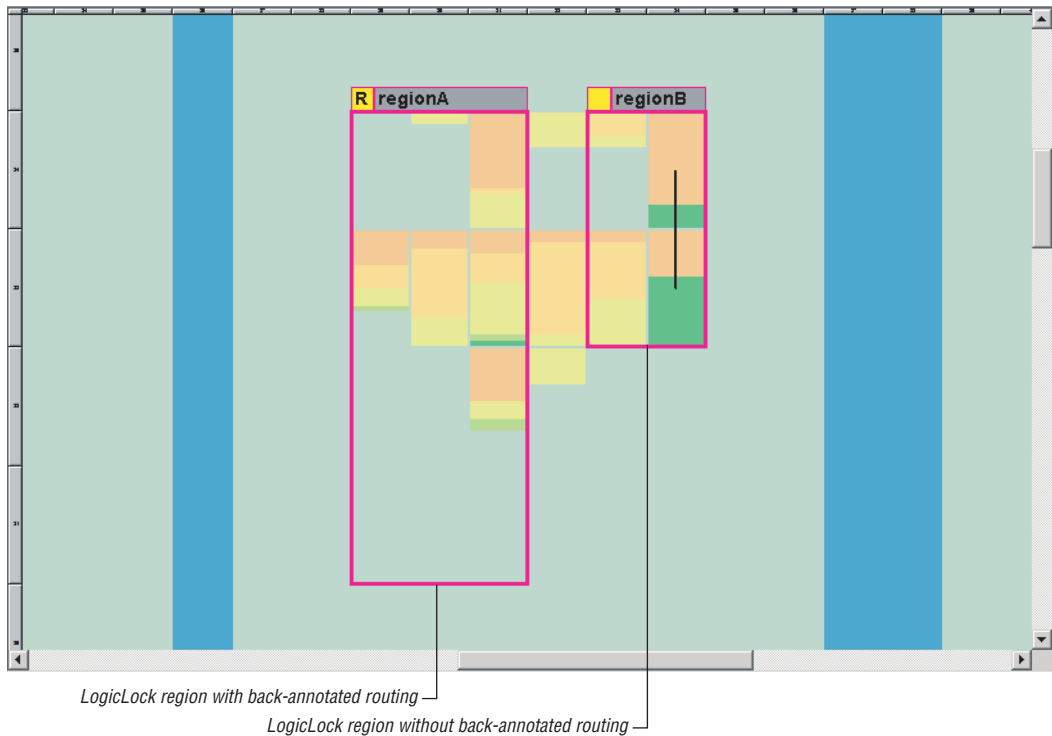
Only regions that have back-annotated routing information have their routing information exported when you export the LogicLock regions. All other regions are exported as regular LogicLock regions.

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”, shown in Figure 10–21.

Figure 10–21. 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 10–22).

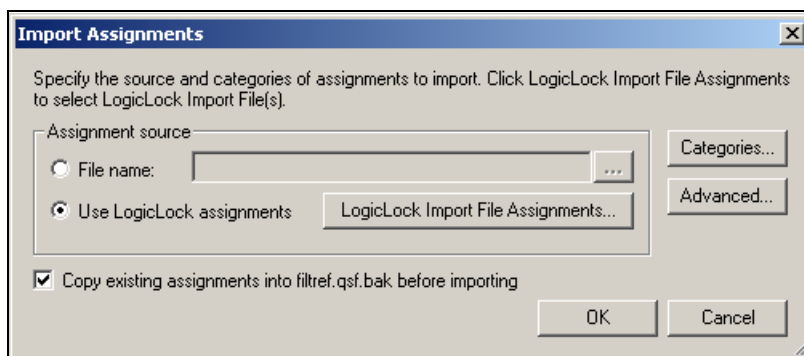
Figure 10–22. Back-Annotation of Routing

Importing Back-Annotated Routing in LogicLock Regions

To import LogicLock region routing information, turn the **Back-annotated routing** option on in the **Advanced Import Assignments** dialog box (Assignments menu). [Figure 10–23](#) 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 defined using the LogicLock **Back-annotated Routing Import File Name** option for the Quartus II software to import routing information for your design.

Figure 10–23. 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, i.e., the VQM, that was used to generate the RCF file. 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 and Uninitializing a LogicLock Region

You must initialize the LogicLock data structures before creating or modifying any LogicLock regions and before executing any of the Tcl commands listed below.

Use the following Tcl command to initialize the LogicLock data structures:

```
initialize_logiclock
```

Use the following command to uninitialize the LogicLock data structures before closing your project:

```
uninitialize_logiclock
```

Creating or Modifying LogicLock Regions

Use the following Tcl command to create or modify a LogicLock region:

```
set_logiclock -auto_size true -floating true -region \  
<my_region-name>
```



In the above example the region's size 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 10–2](#).

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.



For more information about preventing further netlist optimization, refer to [“Prevent Further Netlist Optimization” on page 10–38](#).

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 10–18.

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_name export.qsf
```



For more information about exporting LogicLock Regions see [“Export the Module”](#) on page 10–17.

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
```

```
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 10–19.

Setting LogicLock Assignment Priority

Use the following Tcl code to set the priority for 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_logiclock_member_priority $reverse
```



For more information about Setting the LogicLock Assignment Priority, see [“Constraint Priority”](#) on page 10–30.

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 10–28](#).

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.

Introduction

Timing analysis is performed on an FPGA design to determine that the design's performance meets the required timing goals. This analysis includes system clock frequency (f_{MAX}), setup and hold timing for the design's top-level input ports, as well as clock-to-output timing for all top-level output ports. Measuring these parameters against performance goals ensures that the FPGA design functions as planned in the end target system.

After the FPGA design is stabilized, fully tested in-system, and satisfies the HardCopy® design rules, the design can be migrated to a HardCopy device. Altera® performs the same rigorous timing analysis on the HardCopy device during its implementation, ensuring that it meets the same timing goals. Because the critical timing paths of the HardCopy version of a design are different from the corresponding paths in the FPGA version, meeting the same timing goals is particularly important.

Timing improvements in HardCopy as compared to the equivalent FPGA devices exist for several reasons. While maintaining the same rich set of features as the corresponding FPGA, HardCopy devices have a highly optimized die size to make them as small as possible. Because of the customized interconnect structure that makes this optimization possible, the delay through each signal path is less than the original FPGA design. Quartus® II software versions 4.0 and later determine routing and associated buffer insertion for the design and provides the Timing Analyzer with more accurate information on the delays than was possible in the previous version of the Quartus II software.

The differences in the timing between HardCopy devices and FPGAs is inconsequential as long as the HardCopy device is checked against a specification that fully defines the timing of the design. After this timing goal is fully defined, the HardCopy device is guaranteed to function correctly.

This chapter describes how to meet the required timing performance of HardCopy devices and improve it.

Timing Closure

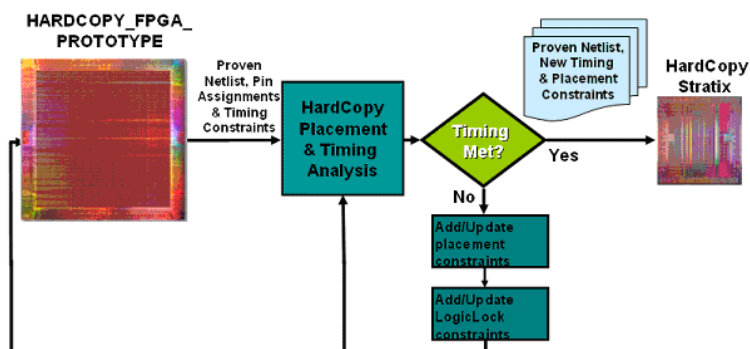
Many of today's developers are faced with the difficult task of meeting the timing goals of systems designed with an ASIC, which can consume many valuable months of intensive engineering effort. The slower development process exists because, in today's silicon technology

(0.18 μm and 0.13 μm), the delay associated with interconnect dominates the delay associated with the transistors used to make the logic gates. Consequently, ASIC performance is sensitive to the physical placement-and-routing of the logic blocks that make up the design.

On migration, the HardCopy device is structurally identical to its FPGA counterpart; there is no re-synthesis or library re-mapping required. Since the interconnect lengths are much smaller in the HardCopy device than they are in the FPGA, the place-and-route engine compiling the HardCopy design has a considerably less difficult task than it does in an equivalent ASIC development. Coupled with detailed timing constraints, the place-and-route is timing driven.

Figure 11–1 illustrates the design flow for estimating performance and optimizing the designs. You can target your designs to `HARDCOPY_FPGA_PROTOTYPE` devices, and pass the design information to the placement and timing analysis engine to estimate the performance of HardCopy Stratix® devices. In the event that the required performance is not met, you can modify or add placement and LogicLock™ constraints. If the performance goals are still not met, then change your RTL source, optimize the FPGA design, and estimate timing iteratively.

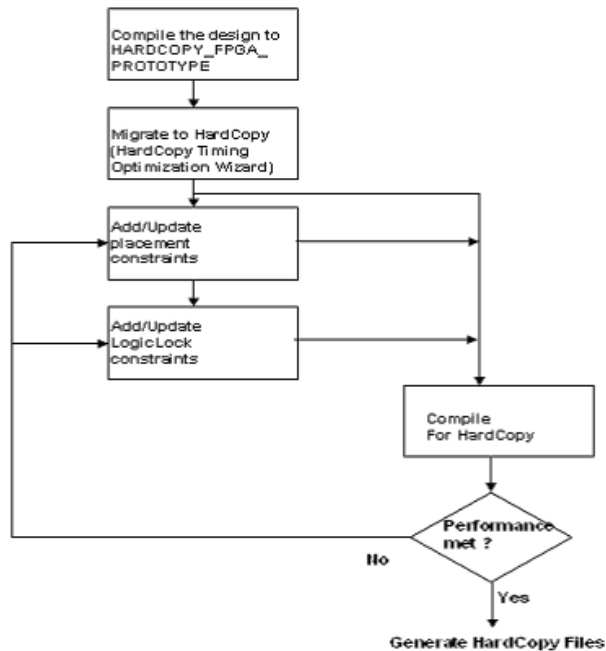
Figure 11–1. Design Flow for Estimating Performance & Optimizing the Design



Placement Constraints

The Quartus II software version 4.0 and later supports placement constraints and LogicLock regions for HardCopy Stratix devices. Figure 11–2 shows an iterative process to modify the placement constraints until the best placement for the HardCopy Stratix device is obtained to achieve the best performance.

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



Location Constraints

Location Array Block (LAB) Assignments

Location constraints for HardCopy Stratix devices are supported. To achieve better performance, you can make LAB-level assignments after migrating the HARDCOPY_FPGA_PROTOTYPE project, and before compiling the design for a HardCopy Stratix device. One important consideration for LAB reassignments is that the entire contents of a LAB is moved to another empty LAB. If you want to move the logic contents of LAB "A" to LAB "B," the entire contents of LAB A is 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.

LogicLock Assignments

LogicLock

Quartus II software enables a block-based design approach using LogicLock. With LogicLock, designers can create and implement each logic module independently, and then integrate all of the optimized modules into the top-level design.



For more information about the LogicLock design methodology, see the *LogicLock Design Methodology* chapter in Volume 2 of the *Quartus II Handbook*.

LogicLock constraints are supported when you are migrating 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 ["Examples of Supported LogicLock Constraints"](#). This modification is necessary because the floorplan of a HardCopy Stratix device is different from that of an equivalent Stratix device. If this modification did not occur, LogicLock assignments would lead to no-fits due to bad placement. Making the regions auto-size and floating maintains your module or entity LogicLock assignments, allowing you to easily adjust the LogicLock regions as required to improve the performance.

Examples of Supported LogicLock Constraints

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
```

Tutorial

To learn more about the LAB and LogicLock assignments, perform the tutorial available on www.altera.com/literature. The tutorial illustrates the performance improvement by LAB and LogicLock assignments. To know more about the performance improvements in general for FPGA designs, refer to the following application notes:

Design Optimization for Altera Devices

http://www.altera.com/literature/hb/qts/qts_qii52005.pdf

Timing Closure Floorplan

http://www.altera.com/literature/hb/qts/qts_qii52006.pdf

LogicLock Design Methodology

http://www.altera.com/literature/hb/qts/qts_qii52009.pdf

Minimizing Clock Skew

The clock is an important component that affects design performance in any digital integrated circuit. The discussion in the remainder of this section pertains to HardCopy APEX™ 20KE, HardCopy APEX 20KC, and HardCopy Stratix devices.

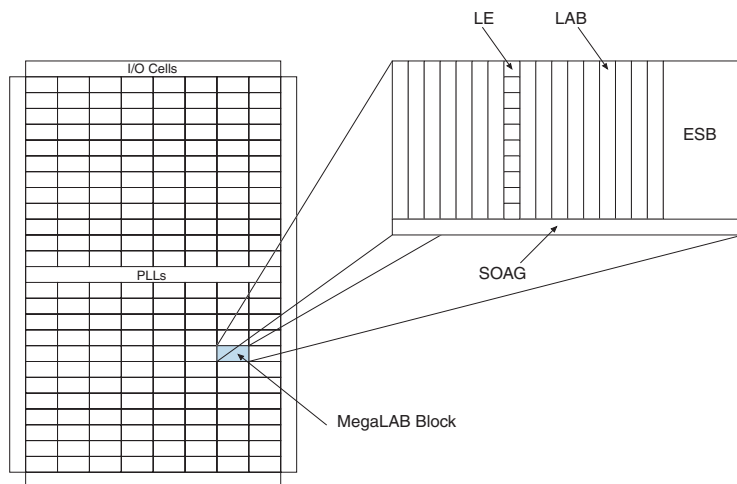
The architecture of the HardCopy HC20K device is based on the APEX 20KE and APEX 20KC FPGA devices and the HardCopy Stratix devices are based on the Stratix FPGA devices. The same dedicated clock trees (CLK[3..0]) that exist in APEX 20KE and APEX 20KC devices or (CLK[15..0]) that exist in Stratix devices also exist in the corresponding HardCopy device. These clock trees are carefully designed and optimized to minimize the clock skew over the entire device. The clock trees are balanced by maintaining the same loading at the end of each point of the clock trees, regardless of what resources (logic elements [LEs], embedded system blocks [ESBs], and input/output elements [IOEs]) are used in any design. The insertion delay of the HardCopy-dedicated clock trees is marginally faster than in the corresponding APEX 20KE, APEX 20KC, or Stratix FPGA device because of the smaller footprint of the HardCopy silicon.

Because there is a large area overhead for these global signals that may not be used on every design, the FAST bidirectional pins (FAST[3..0]) of the HardCopy APEX 20KE and HardCopy APEX 20KC or the dedicated fast regional I/O pins of HardCopy Stratix do not have dedicated pre-built clock or buffer trees in HardCopy devices. If any of the FAST/dedicated fast regional signals are used as clocks, a clock tree is synthesized by the place-and-route tool after the placement of the design has occurred. The skew and insertion delay of these synthesized clock trees are carefully controlled, ensuring that the timing requirements of the

design are met. You can also use the FAST signals of HardCopy APEX or the dedicated fast regional I/O pins of HardCopy Stratix as high fan-out reset or enable signals. For these cases, skew is usually less important than insertion delay. To reiterate, a buffer tree is synthesized after the design placement.

The clock or buffer trees that are synthesized for the FAST pins of HardCopy APEX 20KE and HardCopy APEX 20KC or the dedicated fast regional I/O pins of HardCopy Stratix are built from special cells in the HardCopy base design. These cells do not exist in the FPGA. They are used in the HardCopy design exclusively to meet timing and testing goals. They are not available to make any logical changes to the design as implemented in the FPGA. These resources are called the strip of auxiliary gates (SOAG). There is one of these strips per MegaLAB™ structure in HardCopy devices. Each SOAG consists of a number of primitive cells, and there are approximately 10 SOAG primitive cells per logic array block (LAB). Several SOAG primitives can be combined to form more complex logic, but the majority of SOAG resources are used for buffer tree, clock tree, and delay cell generation. Figure 11–3 shows the SOAG architectural feature.

Figure 11–3. SOAG Architectural Feature



For detailed information on the HardCopy device architecture, including SOAG resources, see the *HardCopy APEX 20K Device Family Data Sheet* chapter in Volume 1 of the *HardCopy Device Handbook*.

Checking the HardCopy Device Timing

To ensure that the timing of the HardCopy device meets performance goals, detailed static timing analysis must be run on the HardCopy design database. For this timing analysis to be meaningful, all timing constraints and timing exceptions that were applied to the design for the FPGA implementation must also be used for the HardCopy implementation. If no timing constraints, or only partial timing constraints, were used for the FPGA design, a full set of constraints must be specified for the HardCopy design by filling in the unspecified constraints with default values. If this is not done, there is no way of knowing if the HardCopy device meets the required timing of the end target system. The timing constraints can be captured through the Timing Wizard in the Altera Quartus II software. The following constraints must be included:

- Clock Definitions
- Primary Input Pin Timing
- Primary Output Pin Timing
- Combinatorial Timing
- Timing Exceptions

Clock Definitions

These definitions are used to describe the parameters of all different clock domains in a design. Clock parameters that must be defined are frequency, time at which the clock edge rises, time at which the clock edge falls, clock uncertainty (or skew), and clock name. Figures 11–4 and 11–5 show these clock attributes.

Figure 11–4. Clock Attributes

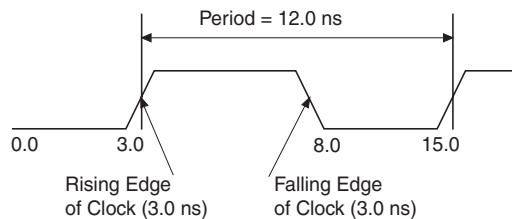
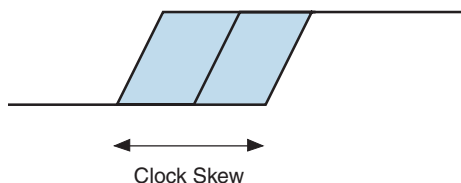
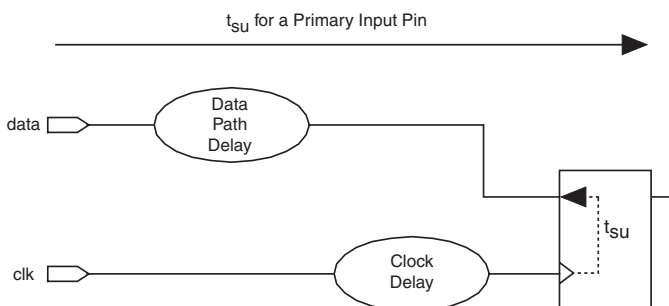


Figure 11-5. Clock Skew

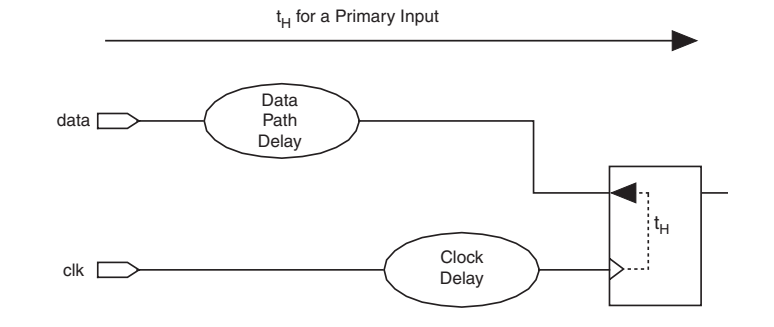
Primary Input Pin Timing

This constraint must be specified for every primary input pin in the design (and for the input path of every bidirectional pin). The input pin timing can be captured in two ways. The first is to describe what maximum on-chip delay is acceptable (i.e., the setup time of a primary input to any register in the design relative to a specific clock). [Figure 11-6](#) depicts a generic circuit with an on-chip setup-time constraint, which may be different for each clock domain.

Figure 11-6. On-Chip Setup-Time Constraint

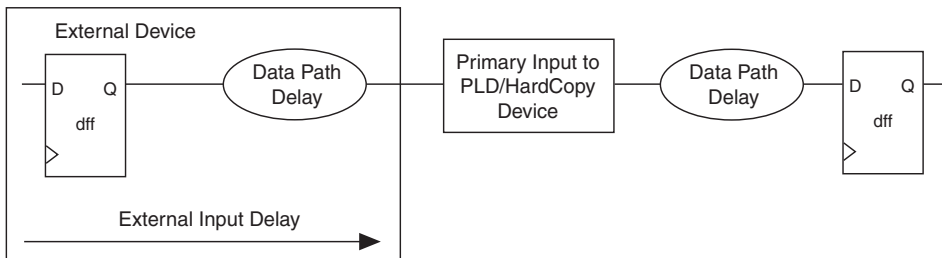
The minimum on-chip delay from any primary input pin must be specified to describe input hold-time requirements. [Figure 11-7](#) depicts a generic circuit with an on-chip hold-time constraint.

Figure 11–7. On-Chip Hold-Time Constraint



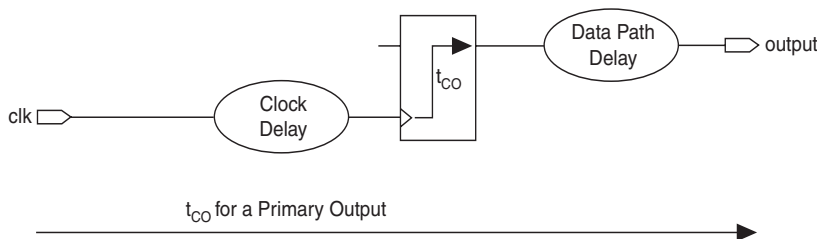
The second way to capture the input pin timing is to describe the external timing environment, which is the maximum and minimum arrival times of the external signals that drive the primary input pins of the HardCopy device or FPGA. Figure 11–8 shows the external timing constraint that drives the primary input pin. This external input delay time can be used by the static timing analysis tool to check that there is enough time for the data to propagate to the internal nodes of the device. If there is not enough time, then a timing violation occurs.

Figure 11–8. External Timing Constraint Driving a Primary Input Pin

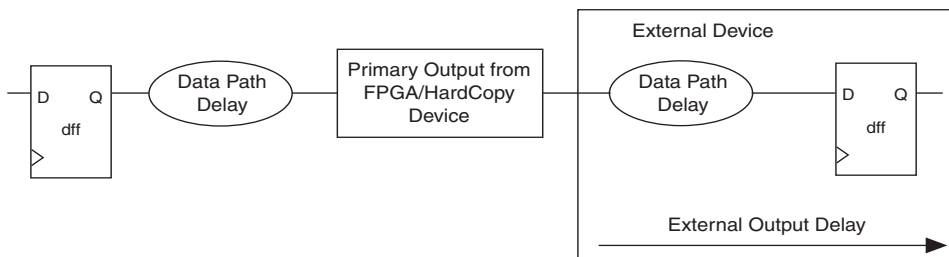


Primary Output Pin Timing

This constraint must be specified for every primary output pin in the design (and for the output path of every bidirectional pin). The output pin timing is captured in two ways. The first is to describe what maximum (and minimum) on-chip clock-to-output (t_{CO}) delay is acceptable (i.e., the time it takes from the active edge of the clock to the data arriving at the primary output pin). Figure 11–9 depicts a generic circuit with an on-chip t_{CO} time constraint. Also, there can be a minimum t_{CO} requirement.

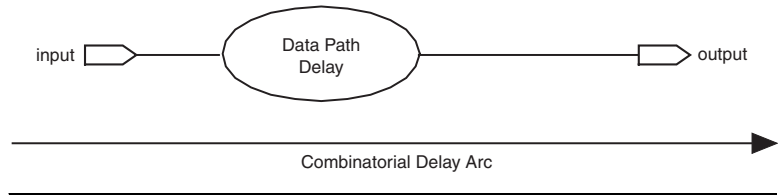
Figure 11–9. On-Chip Clock-to-Output (T_{CO}) Time Constraint

The second way to capture output pin timing is to describe the external timing environment, which is the maximum and minimum delay times of external signals that are driven by the primary output pins of the HardCopy device or FPGA. Figure 11–10 shows the external timing constraint driven by the primary output pin. The static timing analysis tool uses this information to check that the on-chip timing of the output signals is within the desired specification.

Figure 11–10. External Timing Constraint for a Primary Output Pin

Combinatorial Timing

Combinatorial timing occurs when there is a path from a primary input pin to a primary output pin. This type of circuit does not contain any registers. Therefore, it does not require a clock for constraint specification. The maximum and minimum delay from the primary input pin to the primary output pin is all that is needed. Figure 11–11 shows a generic circuit where a combinatorial timing arc constraint must be placed.

Figure 11–11. Combinatorial Timing Constraint

Timing Exceptions

Some circuit structures warrant special consideration. For example, when a design has more than one clock domain and the clock domains are not related, all timing paths between the two clock domains can be ignored. All timing paths using the static timing analysis tool can be ignored by specifying false paths for all signals that go from one clock domain to the other clock domain(s). Additionally, there are circuits that are not intended to operate in a single-clock cycle. These circuits require that you specify multi-cycle clock exceptions.

After the information is captured, it can be used by Altera to directly check all timing of the HardCopy device before tape out occurs. If any timing violations are found in the HardCopy device due to over-aggressive timing constraints, they must either be fixed by Altera, or waived by the customer.



For more information on timing analysis, see the *Quartus II Timing Analysis* chapter and the *Synopsys PrimeTime Support* chapter in Volume 3 of the *Quartus II Handbook*.

Correcting Timing Violations

After the customized metal interconnect is generated for the HardCopy device, Altera checks the timing of the design with an industry standard static timing analysis tool, PrimeTime. Timing violations are reported by this tool, and they are subsequently corrected.

Hold-Time Violations

Because the interconnect in a HardCopy device is customized for a particular application, it is possible that hold-time (t_H) violations exist in the HardCopy device after place-and-route occurs. A hold violation exists if the sum of the delay in the clock path between two registers plus the micro hold time of the destination register is greater than the delay of the data path from the source register to the destination register. The following equation describes this relationship.

$$t_H \text{ Slack} = \text{Data Delay} - \text{Clock Delay} - \text{Micro } t_H$$

If a negative slack value exists, there is a hold-time violation. Any hold-time violation present in the HardCopy design database after the interconnect data is generated is removed by inserting the appropriate delay in the data path. The inserted delay is large enough to guarantee no hold violations under fast, low-temperature, high-voltage conditions.

Table 11–1 shows an example report of a Synopsys PrimeTime static timing analysis of a typical HardCopy design. This report shows that the circuit has a hold-time violation and a negative slack value. Table 11–2 shows the timing report for the same path after the hold violation has been fixed. The instance and cell names shown in these reports are generated as part of the HardCopy implementation process, and are based on the physical location of those elements in the device.

Table 11–1. Static Timing Analysis Before Hold-Time Violation Fix (Part 1 of 2)

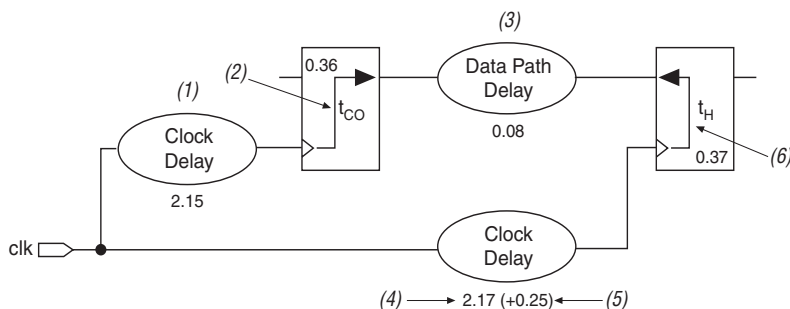
Startpoint:GR23_GC0_L19_LE1/um6 (falling edge-triggered flip-flop clocked by CLK0')			
Endpoint: GR23_GC0_L20_LE8/um6 (falling edge-triggered flip-flop clocked by CLK0')			
Path Group: CLK0			
Path Type:min			
Point	Incr	Path	Reference to Figure 11-12
clock CLK0' (fall edge)	0.00	0.00	
clock network delay (propagated)	2.15	2.15	(1)
GR23_GC0_L19_LE1/um6/clk (c1110)	0.00	2.15 f	(2)
GR23_GC0_L19_LE1/um6/regout (c1110)	0.36 *	2.52 r	(2)
GR23_GC0_L19_LE1/REGOUT (c1000_2d7a8)	0.00	2.52 r	(2)
GR23_GC0_L20_LE8/LUTD (c1000_56502)	0.00	2.52 r	(3)
GR23_GC0_L20_LE8/um1/datad (indsim)	0.01 *	2.52 r	(3)
GR23_GC0_L20_LE8/um1/ndsim (indsim)	0.01 *	2.53 f	(3)
GR23_GC0_L20_LE8/um5/ndsim (mxcascout)	0.00 *	2.53 f	(3)
GR23_GC0_L20_LE8/um5/cascout (mxcascout)	0.06 *	2.59 f	(3)
GR23_GC0_L20_LE8/um6/dcout (c1110)	0.00 *	2.59 f	(3)
data arrival time		2.59	
clock CLK0' (fall edge)	0.00	0.00	
clock network delay (propagated)	2.17	2.17	(4)
clock uncertainty	0.25	2.42	(5)

Table 11–1. Static Timing Analysis Before Hold-Time Violation Fix (Part 2 of 2)

Startpoint: GR23_GC0_L19_LE1/um6
 (falling edge-triggered flip-flop clocked by CLK0')
 Endpoint: GR23_GC0_L20_LE8/um6
 (falling edge-triggered flip-flop clocked by CLK0')
 Path Group: CLK0
 Path Type: min

Point	Incr	Path	Reference to Figure 11-12
GR23_GC0_L20_LE8/um6/clk (c1110)		2.42 f	(6)
library hold time	0.37 *	2.79	
data required time		2.79	
data required time		2.79	
data arrival time		-2.59	
slack (VIOLATED)		-0.20	

Figure 11–12 shows the circuit described by the Table 11–1 static timing analysis report.

Figure 11–12. Circuit with a Hold-Time Violation

Placing the values from the static timing analysis report into the hold-time slack equation results in the following:

$$t_H \text{ Slack} = \text{Data Delay} - \text{Clock Delay} - \text{Micro } t_H$$

$$t_H \text{ Slack} = (2.15 + 0.36 + 0.08) - (2.17 + 0.25) - 0.37$$

$$t_H \text{ Slack} = -0.20 \text{ ns}$$

This result shows that there is negative slack in this path, meaning that there is a hold-time violation of 0.20 ns.

After fixing the hold violation, the timing report for the same path is regenerated (see [Table 11–2](#)). The netlist changes are in *bold italic* type.

Table 11–2. Static Timing Analysis After Hold-Time Violation Fix (Part 1 of 2)

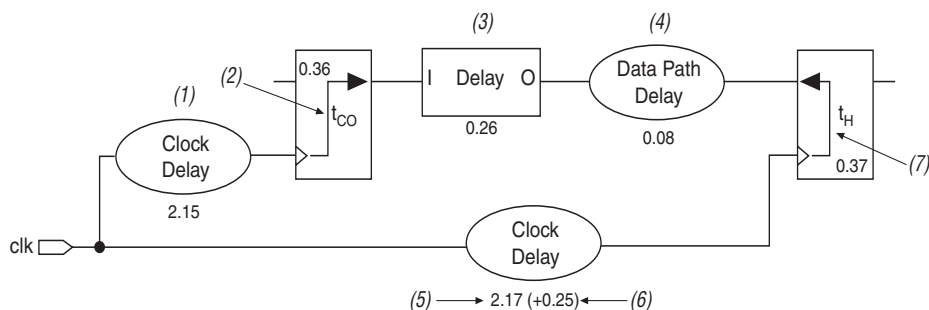
Startpoint: GR23_GC0_L19_LE1/um6
(falling edge-triggered flip-flop clocked by CLK0')
Endpoint: GR23_GC0_L20_LE8/um6
(falling edge-triggered flip-flop clocked by CLK0')
Path Group: CLK0
Path Type: min
Static Timing Analysis After Hold-Time Violation Fix

Point	Incr	Path	Reference to Figure 11–13
clock CLK0' (fall edge)	0.00	0.00	(1)
clock network delay (propagated)	2.15	2.15	(1)
GR23_GC0_L19_LE1/um6/clk (c1110)	0.00	2.15 f	(2)
GR23_GC0_L19_LE1/um6/regout (c1110)	0.36 *	2.52 r	(2)
GR23_GC0_L19_LE1/REGOUT (c1000_2d7a8)	0.00	2.52 r	(2)
<i>thc_916/A (de105)</i>	<i>0.01 *</i>	<i>2.52 r</i>	(3)
<i>thc_916/Z (de105)</i>	<i>0.25 *</i>	<i>2.78 r</i>	(3)
GR23_GC0_L20_LE8/LUTD (c1000_56502)	0.00	2.78 r	(3)
GR23_GC0_L20_LE8/um1/datad (indsim)	0.01 *	2.78 r	(3)
GR23_GC0_L20_LE8/um1/ndsim (indsim)	0.01 *	2.79 f	(3)
GR23_GC0_L20_LE8/um5/ndsim (mxcascout)	0.00 *	2.79 f	(3)
GR23_GC0_L20_LE8/um5/cascout (mxcascout)	0.06 *	2.85 f	(3)
GR23_GC0_L20_LE8/um6/dcout (c1110)	0.00 *	2.85 f	(3)
data arrival time		2.85	
clock CLK0' (fall edge)	0.00	0.00	
clock network delay (propagated)	2.17	2.17	(4)
clock uncertainty	0.25	2.42	(5)
GR23_GC0_L20_LE8/um6/clk (c1110)		2.42 f	(6)
library hold time	0.37 *	2.79	
data required time		2.79	

Table 11–2. Static Timing Analysis After Hold-Time Violation Fix (Part 2 of 2)

data required time	2.79
data arrival time	-2.85
slack (MET)	+0.06

Figure 11–13 shows the circuit described by the Table 11–2 static timing analysis report.

Figure 11–13. Circuit Including a Fixed Hold-Time Violation

Placing the values from the static timing analysis report into the hold-time slack equation results in the following.

$$t_H \text{ Slack} = \text{Data Delay} - \text{Clock Delay} - \text{Micro } t_H$$

$$t_H \text{ Slack} = (2.15 + 0.36 + 0.26 + 0.08) - (2.17 + 0.25) - 0.37$$

$$t_H \text{ Slack} = +0.06 \text{ ns}$$

In this timing report, the slack of this path is reported as 0.06 ns. Therefore, this path does not have a hold-time violation. The path was fixed by the insertion of a delay cell (dcl05) into the data path, which starts at the REGOUT pin of cell GR23_GC0_L19_LE1 and finishes at the LUTD input of cell GR23_GC0_L20_LE8. The instance name of the delay cell in this case is thc_916.



A clock_uncertainty of 0.25 ns is specified in this timing report, and is used to add extra margin during the hold-time calculation, making the design more robust. This feature is a part of the static timing analysis tool, not of the HardCopy design.

The delay cell is created out of the SOAG resources that exist in the HardCopy base design.

Setup-Time Violations

A setup violation exists if the sum of the delay in the data path between two registers plus the micro setup time (t_{SU}) of the destination register is greater than the sum of the clock period and the clock delay at the destination register. The following equation describes this relationship:

$$t_{SU} \text{ Slack} = \text{Clock Period} + \text{Clock Delay} - (\text{Data Delay} + \text{Micro } t_{SU})$$

If there is a negative slack value, it means that there is a setup-time violation. There are several potential mechanisms that can cause a setup-time violation. The first is when the synthesis tool is unable to meet the required timing goals. However, a HardCopy design does not rely on any resynthesis to a new cell library; the synthesis results that were generated as part of the original FPGA design are maintained, meaning that the HardCopy implementation of a design uses exactly the same structural netlist as its FPGA counterpart. For example, if you used a particular synthesis option to ensure that a particular path only contained a certain number of logic levels, the HardCopy design will contain exactly the same number of logic levels for that path. Consequently, if the FPGA was free of setup-time violations, no setup-time violations occur in the HardCopy device as a result of the netlist structure.

The second mechanism that can cause setup-time violations is differing placement of the resources in the netlist for the HardCopy device compared to the original FPGA. This scenario is extremely unlikely as the place-and-route tool used during the HardCopy implementation performs timing-driven placement. In extreme cases, some manual placement modification might be necessary. The placement is performed at the LAB and ESB level, meaning that the placement of logic cells inside each LAB is fixed, and is identical to the placement of the FPGA. IOEs have fixed placement to maintain the pin and package compatibility of the original FPGA.

The third, and most likely, mechanism for setup-time violations occurring in the HardCopy device is a signal with a high fan-out. In the FPGA, high fan-out signals are buffered by large drivers that are an integral part of the programmable interconnect structure. Consequently, a signal that was fast in the FPGA can be initially slower in the HardCopy version, which is before any buffering is inserted into the HardCopy design to increase the speed of the slow signal. The place-and-route tool detects these

signals and automatically creates buffer trees using SOAG resources, ensuring that the heavily loaded, high fan-out signal is fast enough to meet performance requirements.

Table 11–3 shows the timing report for a path that contains a high fan-out signal *before* the place-and-route process. Table 11–4 shows the timing report for a path that contains a high fan-out signal *after* the place-and-route process. Before the place-and-route process, there is a large delay on the high fan-out net that is driven by the pin GR12_GC0_L2_LE4/REGOUT. This delay is due to the large capacitive load that the pin has to drive. For more information on this timing report, see Figure 11–14.

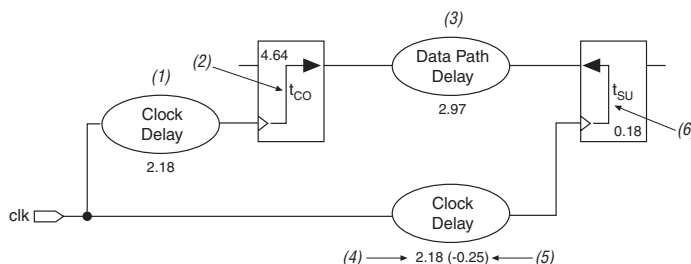
Table 11–3. Timing Report Before the Place-&Route Process (Part 1 of 2)

Startpoint: GR12_GC0_L2_LE4/um6 (falling edge-triggered flip-flop clocked by clkx')			
Endpoint: GR4_GC0_L5_LE2/um6 (falling edge-triggered flip-flop clocked by clkx')			
Path Group: clkx			
Path Type: max			
Point	Incr	Path	Reference to Figure 11–14
clock clkx' (fall edge)	0.00	0.00	(1)
clock network delay (propagated)	2.18	2.18	(1)
GR12_GC0_L2_LE4/um6/clk (c1110)	0.00	2.18 f	(2)
GR12_GC0_L2_LE4/um6/regout (c1110)			(2)
GR12_GC0_L2_LE4/REGOUT (c1000_7f802) <-			(2)
GR4_GC0_L5_LE0/LUTC (c1000_0029a)			(3)
GR4_GC0_L5_LE0/um4/lbt (lt53b)	2.36	9.18 f	(3)
GR4_GC0_L5_LE0/um5/cascout (mxcascout)	0.07	9.24 f	(3)
GR4_GC0_L5_LE0/um2/COMBOUT (icombout)	0.09	9.34 r	(3)
GR4_GC0_L5_LE0/COMBOUT (c1000_0029a)	0.00	9.34 r	(3)
GR4_GC0_L5_LE2/LUTC (c1000_0381a)	0.00	9.34 r	(3)
GR4_GC0_L5_LE2/um4/lbt (lt03b)	0.40	9.73 r	(3)
GR4_GC0_L5_LE2/um5/cascout (mxcascout)	0.05	9.78 r	(3)
GR4_GC0_L5_LE2/um6/dcout (c1110)	0.00	9.78 r	(3)
data arrival time		9.79	(3)
clock clkx' (fall edge)	7.41	7.41	

Table 11–3. Timing Report Before the Place-&Route Process (Part 2 of 2)

clock network delay (propagated)	2.18	9.59	(4)
clock uncertainty	-0.25	9.34	(5)
GR4_GC0_L5_LE2/um6/clk (c1110)		9.34 f	
Point	Incr	Path	Reference to Figure 11–14
library setup time	-0.18	9.16	(6)
data required time		9.16	
data required time		9.16	
data arrival time		-9.79	
slack (VIOLATED)		-0.63	

Figure 11–14 shows the circuit described by the Table 11–3 static timing analysis report.

Figure 11–14. Circuit that has a Setup-Time Violation

The timing numbers in this report are based on pre-layout estimated delays.

Placing the values from the static timing analysis report into the setup-time slack equation results in the following.

$$t_{SU} \text{ Slack} = \text{Clock Period} + \text{Clock Delay} - (\text{Data Delay} + \text{Micro } t_{SU})$$

$$t_{SU} \text{ Slack} = 7.41 + (2.18 - 0.25) - (2.18 + 4.64 + 2.97 + 0.18)$$

$$t_{SU} \text{ Slack} = -0.63 \text{ ns}$$

This result shows that there is negative slack for this path, meaning that there is a setup-time violation of 0.63 ns.

After place-and-route, a buffer tree is constructed on the high fan-out net and the setup-time violation is fixed. The timing report for the same path is shown in [Table 11–4](#). The changes to the netlist are in *bold italic* type. For more information on this timing report, see [Figure 11–15](#).

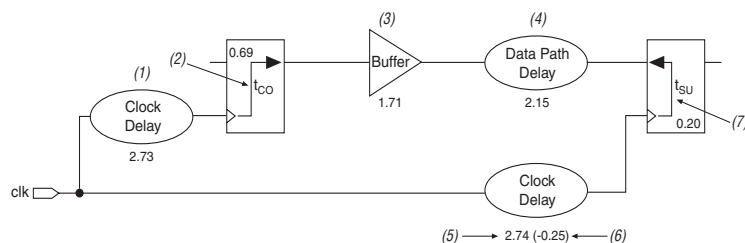
Table 11–4. Timing Report After the Place-and-Route Process (Part 1 of 2)

Startpoint: GR12_GC0_L2_LE4/um6 (falling edge-triggered flip-flop clocked by clkx')			
Endpoint: GR4_GC0_L5_LE2/um6 (falling edge-triggered flip-flop clocked by clkx')			
Path Group: clkx			
Path Type: max			
Point	Incr	Path	Reference to Figure 11–15
clock clkx' (fall edge)	0.00	0.00	
clock network delay (propagated)	2.73	2.73	(1)
GR12_GC0_L2_LE4/um6/clk (c1110)	0.00	2.73 f	(2)
GR12_GC0_L2_LE4/um6/regout (c1110)	0.69 *	3.42 r	(2)
GR12_GC0_L2_LE4/REGOUT (c1000_7f802) <-	0.00	3.42 r	(2)
<i>N1188_iv06_1_0/Z (iv06)</i>	<i>0.06 *</i>	<i>3.49 f</i>	(3)
<i>N1188_iv06_2_0/Z (iv06)</i>	<i>0.19 *</i>	<i>3.68 r</i>	(3)
<i>N1188_iv06_3_0/Z (iv06)</i>	<i>0.12 *</i>	<i>3.80 f</i>	(3)
<i>N1188_iv06_4_0/Z (iv06)</i>	<i>0.10 *</i>	<i>3.90 r</i>	(3)
<i>N1188_iv06_5_0/Z (iv06)</i>	<i>0.08 *</i>	<i>3.97 f</i>	(3)
<i>N1188_iv06_6_2/Z (iv06)</i>	<i>1.16 *</i>	<i>5.13 r</i>	(3)
GR4_GC0_L5_LE0/LUTC (c1000_0029a)	0.00	5.13 r	(4)
GR4_GC0_L5_LE0/um4/ltb (lt53b)	1.55 *	6.68 f	(4)
GR4_GC0_L5_LE0/um5/cascout (mxcascout)	0.06 *	6.74 f	(4)
GR4_GC0_L5_LE0/um2/COMBOUT (icombout)	0.09 *	6.84 r	(4)
GR4_GC0_L5_LE0/COMBOUT (c1000_0029a)	0.00	6.84 r	(4)
GR4_GC0_L5_LE2/LUTC (c1000_0381a)	0.00	6.84 r	(4)
GR4_GC0_L5_LE2/um4/ltb (lt03b)	0.40 *	7.24 r	(4)
GR4_GC0_L5_LE2/um5/cascout (mxcascout)	0.05 *	7.28 r	(4)
GR4_GC0_L5_LE2/um6/dcout (c1110)	0.00 *	7.28 r	(4)
data arrival time		7.28	(4)
Point	Incr	Path	Reference to Figure 11–15

Table 11–4. Timing Report After the Place-and-Route Process (Part 2 of 2)

clock clkx' (fall edge)	7.41	7.41	
clock network delay (propagated)	2.74	10.15	(5)
clock uncertainty	-0.25	9.90	(6)
GR4_GC0_L5_LE2/um6/clk (c1110)		9.90 f	
library setup time	-0.20 *	9.70	(7)
data required time		9.70	
data required time		9.70	
data arrival time		-7.28	
slack (MET)		2.42	

The GR12_GC0_L2_LE4 /REGOUT pin now has the loading on it reduced by the introduction of several levels of buffering (in this case, six levels of inverters). The inverters have instance names similar to N1188_iv06_1_0, and are of type iv06, as shown in the static timing analysis report. As a result, the original setup-time violation of -0.63 ns turned into a slack of +2.42 ns, meaning the setup-time violation is fixed. The circuit that the static timing analysis report shows is illustrated in [Figure 11–15](#). The buffer tree (buffer) is shown as a single cell.

Figure 11–15. Circuit Post Place-&-Route

Placing the values from the static timing analysis report into the setup-time slack equation results in the following:

$$t_{SU} \text{ Slack} = \text{Clock Period} + \text{Clock Delay} - (\text{Data Delay} + \text{Micro } t_{SU})$$

$$t_{SU} \text{ Slack} = 7.41 + (2.74 - 0.25) - (2.73 + 0.69 + 1.71 + 2.15 + 0.20)$$

$$t_{SU} \text{ Slack} = +2.42 \text{ ns}$$

This result shows that there is positive slack for this path, meaning that there is now no setup-time violation.

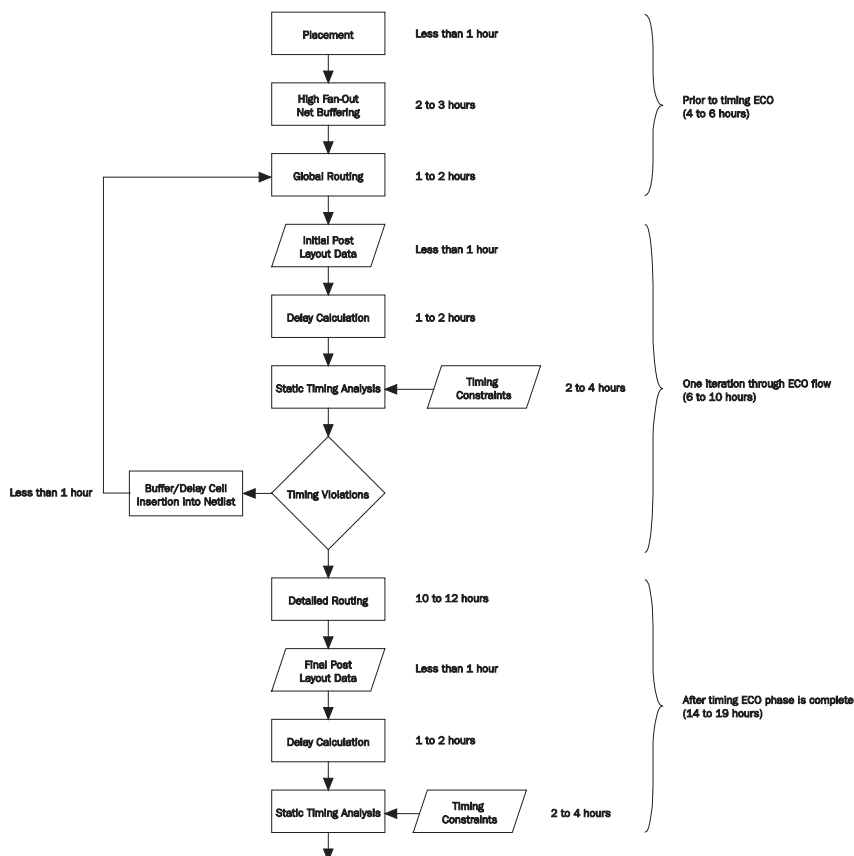
Timing ECOs

In an ASIC, small incremental changes to a design database are termed engineering change orders (ECOs). In the HardCopy design flow, ECOs are performed after the initial post-layout timing data is available.

Static timing analysis is run on the design and a list of paths with timing violations are generated. The netlist is then automatically updated with changes that correct these timing violations (i.e., the addition of delay cells to fix hold-time violations). After the netlist update, the place-and-route database is updated to reflect the netlist changes. The impact on this database is minimized by maintaining all of the pre-existing placement-and-routing, and only changing the routing where new cells are inserted.

The parasitic (undesirable, but unavoidable) resistances and capacitances of the customized interconnect are extracted and then used in conjunction with the static timing analysis tool to re-check the timing of the design. Only a single iteration of this process is typically required to fix all timing violations. The entire ECO stage takes less than a day to complete.

[Figure 11-16](#) shows this flow in more detail, along with the typical duration of each stage.

Figure 11–16. ECO Flow Diagram

Conclusion

When migrating a design from an FPGA implementation to a HardCopy implementation, it is critical to maintain performance even though all timing within the design does not remain exactly the same. These timing differences are inevitable. However, they are rendered inconsequential to the device's behavior in the end-system environment if the HardCopy device meets the system timing constraints. As a standard and automated part of the HardCopy design conversion process, this rendering is achieved through the exhaustive timing analysis that the design undergoes in conjunction with sophisticated timing-driven place-and-route. Static timing analysis can reveal timing violations that are then fixed automatically as part of the HardCopy design process.

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.

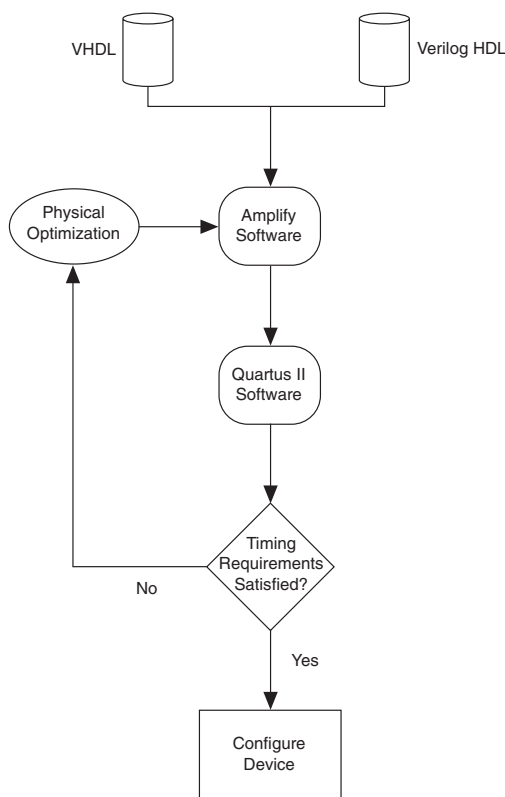


See [“Forward Annotating Amplify Physical Optimizer Constraints into the Quartus II Software”](#) on page 12–12 for more information on setting up a Quartus II project with Amplify Physical Optimizer Tcl script files.

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 12–1](#) shows the Amplify Physical Optimizer design flow.

Figure 12–1. 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.

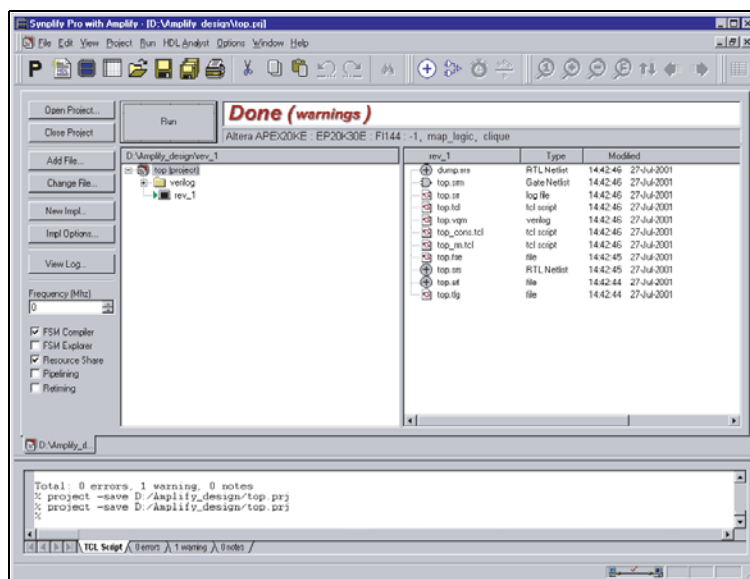
Create New Implementations

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 Clipping** 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 12–2](#) 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 12–2. 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.

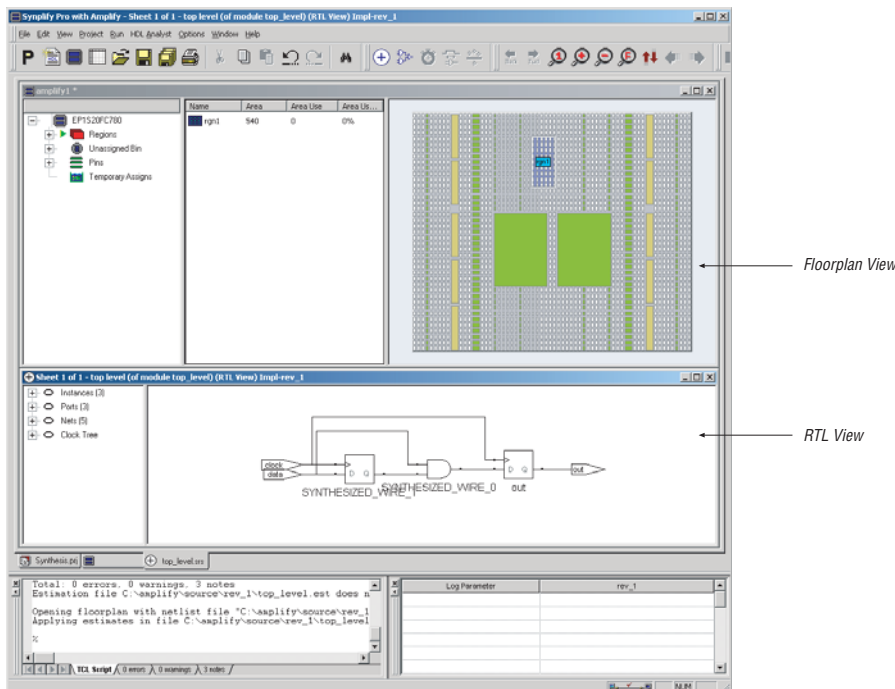
You must determine which paths to constrain based on the critical paths from the previous implementation. When Quartus II projects are launched with the Amplify Tcl script, the Quartus II software generates a `<project name>.tan.rpt` file that lists the critical paths for the design. You

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 12–3).

Figure 12–3. 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:

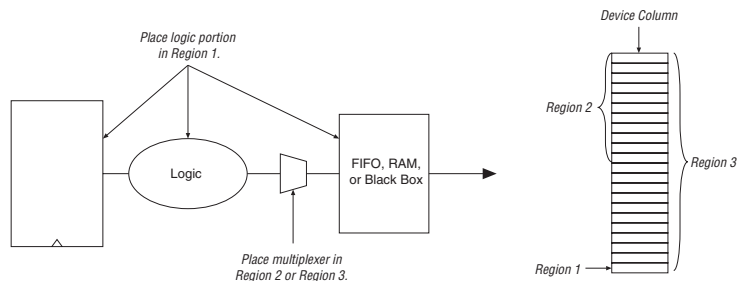
1. Create a region in the Amplify Physical Optimizer device floorplan window and select the module in the RTL view of the design.
2. Drag the module to the new region. The software will then report the utilization of the region.
3. Resynthesize the design in the software to reoptimize the critical path 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.

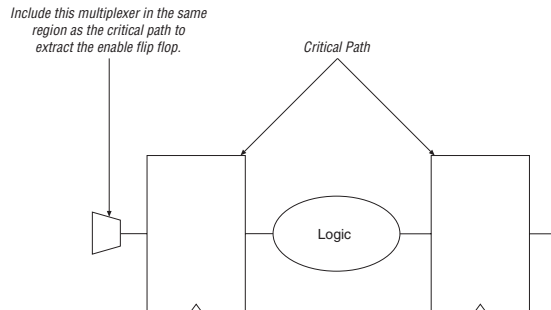
Multiplexers

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 12–4](#) shows placing critical paths with multiplexers.

Figure 12–4. Placing Critical Paths with Multiplexers

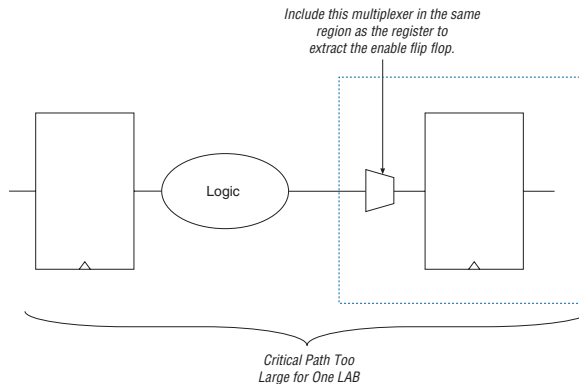


If the critical path contains a multiplexer feeding a register, create a region and place the multiplexer along with the entire critical path in the region. See [Figure 12–5](#).

Figure 12–5. 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 12–6 shows large critical paths with multiplexers feeding registers.

Figure 12–6. 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.

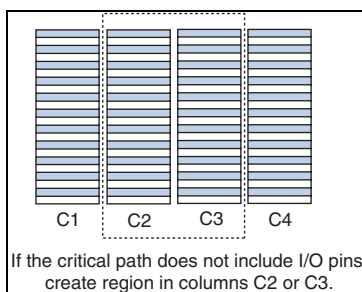
3. Assign the second critical path to the second region.

Feedback Paths

If critical paths have the same start and end points, follow the steps below in the Amplify Physical Optimizer software (see [Figure 12–7](#)):

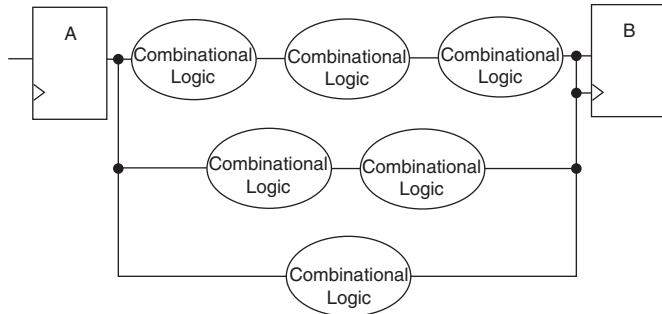
1. Select the register and instance not directly connected to the register.
2. Select **Filter Schematic** twice (right-click menu).
3. Highlight the line leading out of the register and either press **P** or right-click the line. Select **Expand Paths**. Assign this logic to a region.

Figure 12–7. Critical Paths with the Same Starting or Ending Points

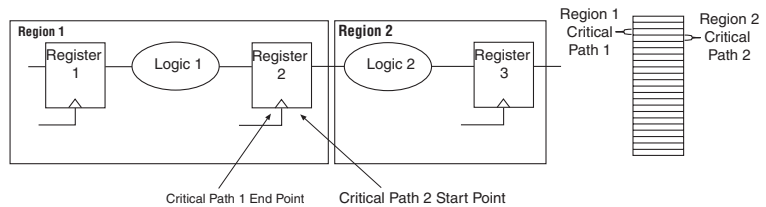


Starting and Ending Points

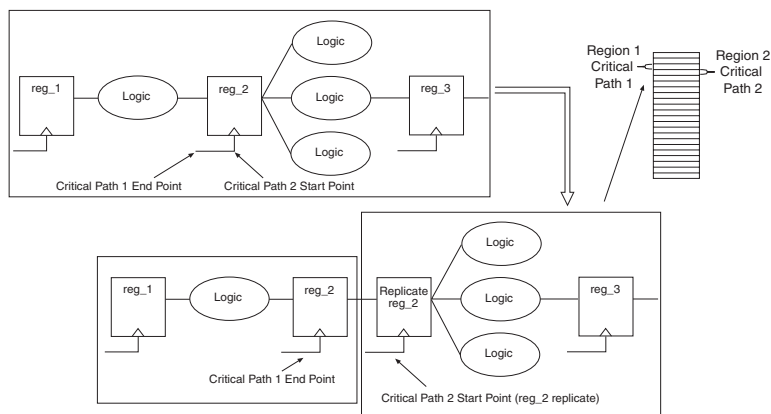
[Figure 12–8](#) shows a critical path that has multiple starting and ending points. Use **Find** to display all the starting and ending points in the RTL view in Amplify. Expand the paths between those points. If there is unrelated logic between the multiple starting points and ending points, assign 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 12–8. 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 12–9](#) shows two critical paths that share a register.

Figure 12–9. 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 12–10](#)). This is done by dragging the same register to the required regions. Entities and nodes are also replicated by performing the same procedure.

Figure 12–10. 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 12–1 provides a description of each script's purpose.

Table 12–1. Amplify Physical Optimizer Tcl Script Description	
Tcl File	Description
<code><project name>_cons</code>	This Tcl script will create and compile a Quartus II project. The <code><project name>.tcl</code> will automatically be sourced when this script is sourced.
<code><project name></code>	This script contains forward annotation of constraint information including clock frequency, duty cycle, location, etc.
<code><project name>_rm</code>	This script removes any previous constraints from the project. The removed constraint is saved in <code><project name>_prev.tcl</code>

To forward annotate Amplify Physical Optimizer's constraints into the Quartus II software you must use **quartus_cmd**. The **quartus_cmd** command must be used as Amplify Physical Optimizer's Tcl scripts are not compatible with **quartus_sh**. The following command will execute the `<project name>_cons`, which will create a Quartus II project with all Amplify Physical Optimizer constraints forward annotated, and will perform a compilation.

```
<command prompt>quartus_cmd 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 will cause 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 12–14.

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 Optimizer Constraints
 - Forward Annotating 12-12
- Physical Synthesis Concepts 12-1
- to-Quartus II Flow 12-2
- Using Amplify Physical Optimizer
 - Floorplans 12-5
- Using the Amplify Physical Optimizer
 - Floorplans 12-5

Arithmetic 3-3

Assignment

- Back-Annotating 8-18
- Location 5-12

Assignment Editor 5-7

- Category 1-2
- Customizable Columns 1-12
- Features 1-3
- Settings Made Outside User Interface 1-1
- Using 1-1
- Using to Place Logic 10-14

Assignments

1-8

- Dynamic Syntax Checking 1-9
- Exporting 1-8, 1-13, 1-14
- Import 1-13, 1-16, 10-20
- Importing 1-16
- Path-Based 10-24
- Revisions 4-1

Atom Netlist

- Design Information 10-18
- Specify 10-19

B

Back-Annotating 10-40

Back-Annotation 10-33

Bar

- Category 1-2
- Edit 1-3
- Information 1-2, 1-6

- Node Filter 1-2, 1-5, 1-10
- Black Box Methodology 12-14

C

Clear Box Methodology 12-13

Clock

- Definitions 11-7
- Minimizing Skew 11-5

Clock Speed

- Improving in Design 6-53

Clocks

- Using Fast Regional in Stratix Devices 6-26

Clock-to-Output Time

- Improving 6-27, 6-51

cmdline Package 3-11

Collection Commands 3-15

Columns

- Customizable 1-7

Combinational

- Timing 11-10

Combinational Logic

- Physical Synthesis 8-10

Command

Line

- Accessing Arguments 3-11
- Options 2-9, 2-11, 3-9
- Scripting Help 2-7

Nested 3-3

Prompt 4-8

Command Prompt 5-12

Compilation

- Initial Settings 6-3
- Restore Original Results 4-5
- Settings
 - Initial 6-2
- Time 6-11
- Timing Driven 6-3, 6-21

Compile Archive 9-14

Compilation

Time

- Optimization Techniques 6–55
- Constraint Priority 10–30
- Constraints
 - Location 11–3
 - Placement 11–3
 - Remove Fitter 6–16
- Control Structures 3–4
- Create New Implementations 12–4
- Critical Path
 - Reducing Delay 6–38
- Custom Region 6–36
- Custom Space 9–11
 - Simple 9–19
 - XML Schema 9–21

D

- Databases
 - Exporting 4–7, 4–10
 - Importing 4–7, 4–10
 - Version-Compatible 4–7, 4–10
- Delays
 - Programmable 6–23
- Design
 - Analysis 6–6
 - Compile 3–13
 - Compile & Verify 10–22
 - Creating Different Versions 4–4
 - File
 - Check Syntax 2–11
 - Fit Quickly 2–13
 - Fit Using Multiple Seeds 2–13
 - Flow 1–10, 2–3, 5–1
 - Complete Design Files 5–4
 - No Design Files 5–2
 - Partial Design Files 5–4
 - Improving Performance 6–30
 - Optimization
 - for Altera Devices 11–5
 - Improve Resource Utilization 6–13
 - Optimization for Altera Devices 6–1
 - Optimize 6–3
 - Optimizing Compilation Time 6–55
 - Using Revisions 4–1
- DESIGNSPACE 9–17
- Device
 - Using Larger 6–20

- Device Resources
 - Reserve 6–42
- Device Settings 6–2
- Drag & Drop
 - Using to Place Logic 10–13
- DSE
 - Advanced Information 9–15
 - Advanced Search Options 9–7
 - Archive Compiles 9–14
 - Command Line Options 9–2
 - Computer Load Sharing 9–15
 - Concepts 9–1
 - Creating Custom Spaces 9–16
 - Distributed Using LSF 9–15
 - Distributed Using Quartus II Master
 - Process 9–15
 - Exploration 9–1, 9–2, 9–6
 - Exploration Settings 9–4
 - Exploration Space 9–8
 - Flow 9–4
 - Flow Options 9–13
 - General Information 9–2
 - LogicLock Region Restructuring 9–8
 - Optimization 9–7
 - Performance Options 9–7
 - Performing Advanced Search 9–7
 - Project Settings 9–6
 - Seed & Seed Sweeping 9–1
 - Support for Altera device Families 9–5
- DSP Blocks
 - Retarget 6–19

E

- EDA
 - Tool Assignments 3–1, 3–18
- Edit Bars 1–2
- Executables
 - Supporting Tcl 2–11, 3–8, 3–9
- Exploration
 - Base Compile Failure 9–13
 - Run Quartus Assembler 9–13
 - Space 9–8
 - Stop Flow After Gain 9–14
 - Stop Flow After Time 9–14
- Exploration Space
 - Area Optimization Search 9–10

- Custom Space 9–11
 - Extra Effort Search 9–9
 - Physical Synthesis Search 9–10
 - Retiming Search 9–10
 - Save to File 9–14
- ## F
- Fan-In
 - Estimating 6–44
 - Fan-Out Control
 - Duplicate Logic 6–32
 - Fast Regional Clocks
 - Using in Stratix Devices 6–26
 - Feedback Paths 12–9
 - Floorplan
 - Timing Closure 7–1
 - Floorplans
 - Detailed 12–11
 - fMAX
 - Improving 6–40
 - Timing Analysis Report 7–19
 - Timing Optimization Techniques 6–27, 6–62
- ## G
- Global Control Signals
 - Dedicated Inputs 6–41
- ## H
- Hierarchy
 - Assignments 5–1, 6–33
 - Flatten 6–31
 - Window 10–7
- ## I
- I/O
 - Assignment
 - Analysis 5–2, 5–13
 - Anaylsis 5–1
 - Anaysis
 - Tcl Command 5–13
 - Creating 5–1, 5–6
 - Design Flow 5–1
 - Inputs Used for Analysis 5–6
 - Placement 5–10
 - Planning 5–1
 - Running Analysis 5–9
 - Understanding Analysis Report 5–9
 - Timing 6–5, 6–7
 - Optimization Techniques 6–21, 6–61
 - Using a PLL to Improve 6–26
 - Incremental Fitting 6–57
 - Independent Paths 12–8
 - Information 1–2
 - Initial Pass
 - No Physical Constraints 12–3
 - Interactive Shell Mode 2–12
 - Iterative Passes
 - Optimizing the Critical Paths 12–5
- ## L
- LCELL Buffers
 - Using to Reduce Required Resources 6–48
 - Lists 3–4
 - Location Assignments 5–12
 - Logic Lock
 - Region
 - Properties 10–10
 - LogicLock 11–4
 - Additional Quartus II Design
 - Features 10–22
 - Assignment Precedence 10–26
 - Assignments 6–32, 11–4
 - Location & Back Annotation 6–35
 - Back Annotation 6–36
 - Constraint Priority 10–30
 - Design Features 10–2
 - Design Flow 10–16, 10–40
 - Design Hierarchy 10–7
 - Design Methodology 10–1, 11–5
 - Drag & Drop 10–13
 - DSE 9–8
 - Examples of Supported Constraints 11–4
 - Importing Functions 3–21
 - Improving Design Performance 10–1
 - Manual Placement 6–36
 - Path Assignments 6–34
 - Region
 - Assigning Content 10–13, 10–37
 - Back-Annotating 10–40

- Exporting 10-33
- Importing 10-35
- Back-Annotating Routing
 - Information 10-33
- Connectivity 7-12, 10-23
- Creating 10-2, 10-37
- Exporting 10-39
- Hierarchical 10-11
- Importing 10-39
- Initializing 10-37
- Modifying 10-37
- Obtaining Properties 10-37
- Placing 10-30
- Placing Memory 10-32
- Placing Other Device Features 10-32
- Placing Pins 10-32
- Prevent Assignment Option 10-23
- Properties 10-2
- Reserve 10-23
- Specify Size and Location 10-10
- Tcl Command 10-8
- Tcl Scripts 10-16
- Uninitializing 10-37
- Viewing Routing Congestion 7-15
- Window 10-2
- Regions 6-57
 - Back Annotating 6-35
 - Back-Annotated 6-59
 - Custom 6-36
 - Hierarchical 6-33
 - Soft LogicLock Regions 10-27
- Regions 5-2
- Restrictions 10-30
- Revisions Feature 10-26
- Setting Assignment Priority 10-39
- Tooltips 10-22

M

- Macrocell Usage
 - Resolving Issues 6-46
- Makefile
 - Implementation 2-14
- Mapped Netlist
 - Generating 5-8, 5-12
- Maximum Frequency
 - Improving 6-53

- Memory Blocks
 - Retarget 6-18
- Messages
 - Error 5-11
 - Status 5-11
- Modular Executables 2-7
 - Makefile 2-14
 - Report Files 2-6
- Module
 - Export 10-17
 - Import 10-19
 - Optimize 10-16
 - Synthesize 10-16
- Module Performance
 - Preserving 10-2
- Multiplexers 12-7

N

- Netlist
 - Optimization 8-1, 10-38
 - Applying Options 8-15
- Node Filter 1-2
- Node-level Netlist
 - Save into Persistent Source File 10-38

O

- Optimization
 - Techniques
 - Resource Utilization 6-13
 - Techniques, LUT-Based Devices 6-12
- Optimization Techniques 8-2
- Optimize Source Code 6-19, 6-39
- Optimize Synthesis for Speed 6-30, 6-31
- Optimizing
 - Critical Path 6-38
 - Critical Paths 12-5
 - Placement
 - Cyclone Devices 6-39
 - Mercury, APEX II, APEX 20KE/C
 - Devices 6-39
 - Stratix Family Devices & Cyclone II
 - Devices 6-38
- Optimization Advisor 6-12
- Output Pins
 - Estimate Fan-In When Assigning 6-44

P

- Parallel Expanders
 - Used Within a LAB 6-45
- PARAM 9-18
- Path
 - Assignments 6-34
 - Feedback 12-9
 - Independent 12-8
- Physical Synthesis
 - Combinational Logic 8-10
 - Optimization 8-17
 - Optimization 6-28
 - Preserving Results 8-14
 - Register Retiming 8-13
 - Report 8-13
 - Search 9-10
- Physical Synthesis Optimization 8-9
- Pin
 - Assignment
 - Output Enable 6-43
 - Assignment
 - Control Signal 6-43
 - Guidelines 5-6
 - Location 5-7
 - Modify 6-20
 - Outputs Using Parallel Expander 6-44
 - Assignment Guidelines & Procedures 6-42
 - Assignments
 - Floorplan Editor 5-8
 - Reserving 5-6, 5-11
 - Timing
 - Primary Input 11-8
 - Primary Output 11-9
- Pin Assignment
 - Minimize Fitting Issues 6-42
- Pipling
 - Complex Register Logic 6-53
- Placement 5-10, 10-18
- Placement Time
 - Reduce Using Incremental Fitting 6-57
- PLL
 - Using to Shift Clock Edges 6-26
- POINT 9-17
- Project Management 4-1
- Projects
 - Archiving 4-5, 4-9
 - Creating 3-12
 - Making Assignments 3-12

- Restoring Archived 3-12, 4-9
- Propagation Delay
 - Improving 6-52

Q

- QFlow Script 2-16
- QSF
 - Specify 10-19
- Quartus II
 - Megafunctions
 - Using MegaWizard Plug-In Manager with Amplify 12-13
 - MegaWizard Plug-In
 - Manager-generated Verilog HDL Files for Clear Box Megafunction
 - Instantiation 12-13
 - MegaWizard Plug-In
 - Manager-generated VHDL Files for Clear Box Megafunction
 - Instantiation 12-13
 - Modular Executables 2-1, 2-2, 2-17
- quartus_sh --flow
 - Compilation 2-7

R

- Register Packing 6-13
- 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 6-22
- Repair Branch 10-22
- Report Data
 - Extracting 3-14
- Reserving Pins 5-11
- Resource Utilization 6-6
 - Analysis & Synthesis by Entity 10-24
 - Optimization Techniques 6-13, 6-41, 6-60
 - Resolving Issues 6-20
 - Resolving Problems 6-45
- Resynthesis
 - Perform WYSIWYG for Area 6-16
 - WYSIWYG Primitive 8-2
- Revision
 - Comparing 4-3

- Creating 4-1, 4-2, 4-8, 4-10
- Deleting 4-2, 4-9
- Getting List 4-9
- Managing 4-8
- Setting Current 4-8
- Routing 10-19
 - Congestion 6-59, 7-15
 - Resolving Issues 6-47
- Routing Time
 - Reducing 6-59
- Rubber Banding 10-23

S

- Scripting Support 3-6, 3-10, 3-13, 4-8, 5-11, 6-59, 8-16, 10-36
- Seed 6-30
 - Sweep 9-9
 - Extra Effort Search 9-9
 - Sweeping 9-1
- Settings 6-30
 - Fitter Effort 6-4, 6-56
 - Initial Compilation 6-60
 - Physical Synthesis Effort 6-56
 - Smart Compilation 6-3
- Setup Time
 - Improving 6-50
- Show
 - Connection Count 10-24
 - Critical Paths 10-24
- Signature Mode 9-12
- Source Code
 - Optimize 6-19, 6-39
 - Optimizing Using Pipelining Technique 6-53
- Spreadsheet 1-2
- Starting and Ending Points 12-9
- State Machine Encoding
 - Change 6-17, 6-31
- Synthesis
 - Netlist Optimization 6-28, 8-2, 8-4
 - Optimize for Area 6-16, 6-17
 - Optimize for Speed 6-31
 - Options, Other 6-32
 - Reduce Netlist Optimization Time 6-55
 - Reducing 6-55
 - Set Effort to High 6-31

- Specify State Machine Encoding 6-17
- Synthesis Netlist
 - Optimization 8-16
 - Preserving Optimization Results 8-8
- Synthesis Netlist Optimization 6-55

T

- TCL
 - Interface 1-13
- Tcl
 - API Reference 2-9, 3-6
 - Assignment 6-60
 - Back-Annotate Command 8-18
 - Command 3-13, 3-15, 5-12
 - Console Window 3-10
 - Defined 3-1
 - Evaluate 2-12, 3-10
 - Getting Help 2-17, 3-25
 - Help 3-2
 - List 3-4
 - Loading Packages 3-8
 - Packages 2-9, 3-6, 3-25
 - Procedure 3-5
 - Quartus II Legacy Support 3-10, 3-28
 - Quartus II Legacy Support 3-2
 - Script
 - Run 2-11, 4-8
 - Scripting Basics 3-2
 - Scripts 3-11, 3-13, 10-8, 10-15
 - Shell in Interactive Mode 3-22
 - Tk
 - GUI Help Interface 3-2, 3-28
- Time Groups
 - Using 1-11
- Timing
 - Checking the HardCopy Device 11-7
 - Correcting Violations 11-11
 - ECOs 11-21
 - Exceptions 11-11
 - fMAX Optimization Techniques 6-27
 - Hold-Time Violations 11-11
 - Improving Propagation Delay 6-52
 - Optimization Techniques 6-12, 6-49
 - Macrocell-Based CPLDs 6-41
 - Setup-Time Violations 11-16
 - Timing Closure 11-1

- Design Analysis 7-23
- Floorplan 5-8, 7-1, 11-5
 - Assigning LogicLock Region
 - Content 10-13
 - Design Analysis 7-1
 - Viewing Assignments 7-3
 - Viewing Critical Paths 7-5
 - Views 7-1
- Floorplan Editor 10-6
 - LogicLock Regions Connectivity 10-23
 - Physical Timing Estimates 7-11
 - View 10-9
- Floorplan Views 7-1
 - in HardCopy Devices 11-1
- Timing Requirements 6-2
- Tooltips 10-22

U

- Using 1-11
- Utilization 6-6, 12-11
 - Resolving Resource Issues 6-20

V

- Variables 3-3
- Verilog HDL Files
 - Clear Box Megafunction Instantiation 12-13
- VHDL Files
 - Clear Box Megafunction Instantiation 12-13
- Virtual Pins 10-28
 - Assigning 10-40



Quartus II Handbook, Volume 3

Verification



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

Copyright © 2004 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



I.S. EN ISO 9001



Chapter Revision Dates	xi
-------------------------------------	-----------

About this Handbook	xiii
----------------------------------	-------------

How to Contact Altera	xiii
-----------------------------	------

Typographic Conventions	xiii
-------------------------------	------

Section I. Simulation

Revision History	Section I-1
------------------------	-------------

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	1-3
Gate-Level Timing Simulation	1-4
Functional RTL Simulation	1-4
Functional RTL Simulation Libraries	1-4
Simulating VHDL Designs	1-5
Simulating Verilog Designs	1-7
Gate-Level Timing Simulation	1-11
Quartus II Software Output Files for use in the ModelSim-Altera Software	1-11
Gate Level Simulation Libraries	1-12
Simulating VHDL Designs	1-15
Simulation Verilog Designs	1-17
Using the NativeLink Feature with ModelSim	1-19
Software Licensing & Licensing Set-Up	1-20
LM_LICENSE_FILE Variable	1-20
Conclusion	1-20

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	2-2
Post-Synthesis Simulation	2-4
Gate-Level Timing Simulation	2-6

Common VCS Compile Switches	2-8
Using VirSim: The VCS Graphical Interface	2-9
VCS Debugging SupportæVCS Command-Line Interface	2-9
Using PLI Routines with the VCS Software	2-10
Preparing & Linking C Programs to Verilog Code	2-10
Scripting Support	2-10
Generating a Post-Synthesis Simulation Netlist for VCS	2-11
Generating a Gate-Level Timing Simulation Netlist for VCS	2-11
Conclusion	2-12

Chapter 3. Cadence NC-Sim Support

Introduction	3-1
Software Requirements	3-1
Simulation Flow Overview	3-1
Functional/RTL Simulation	3-2
Gate-Level Timing Simulation	3-3
Operation Modes	3-3
Quartus II/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	3-10
Compile Source Code & Testbenches	3-11
Elaborate Your Design	3-13
Add Signals to View	3-15
Simulate Your Design	3-17
Gate-Level Timing Simulation	3-18
Quartus II Simulation Output Files	3-18
Quartus II Timing Simulation Libraries	3-20
Set Up Your Environment	3-20
Create Libraries	3-20
Compile the Project Files & Libraries	3-21
Elaborate the Design	3-21
Add Signals to View	3-23
Simulate Your Design	3-23
Incorporating PLI Routines	3-23
Dynamically Link	3-24
Dynamically Load	3-25
Statically Link	3-28
Scripting Support	3-29
Generate NC-Sim Simulation Output Files	3-29
Conclusion	3-30
References	3-30

Section II. Timing Analysis

Revision History	Section II-1
------------------------	--------------

Chapter 4. Quartus II Timing Analysis

Introduction	4-1
Timing Analysis Basics	4-1
Clock Setup Time (tSU)	4-1
Clock Hold Time (tH)	4-2
Clock-to-Output Delay (tCO)	4-3
Pin-to-Pin Delay (tPD)	4-3
Maximum Clock Frequency (fMAX)	4-3
Slack	4-4
Hold Time Slack	4-4
Clock Skew	4-5
Executing Tcl Script-Based Timing Commands	4-6
Setting up the Timing Analyzer	4-6
Setting Global Timing Assignments	4-7
Specifying Individual Clock Requirements	4-7
Setting Other Individual Timing Assignments	4-8
Timing Wizard	4-12
Timing Analysis Reporting in the Quartus II Software	4-12
Advanced Timing Analysis	4-13
Clock Skew	4-13
Multiple Clock Domains	4-15
Multicycle Assignments	4-16
Typical Applications of Multicycle Assignments	4-19
False Paths	4-28
Fixing Hold Time Violations	4-31
Timing Analysis Across Asynchronous Domains	4-32
Minimum Timing Analysis	4-33
Minimum Timing Analysis Settings	4-33
Performing Minimum Timing Analysis	4-33
Minimum Timing Analysis Reporting	4-34
Third-Party Timing Analysis Software	4-34
Advanced Timing Analysis & Reports Using Tcl Scripts	4-34
Conclusion	4-37

Chapter 5. Synopsys PrimeTime Support

Introduction	5-1
Quartus II Settings to Generate PrimeTime Files	5-1
Files Generated for the PrimeTime Environment	5-2
Sample of Constraints Specified in PrimeTime Format	5-4
PrimeTime Timing Reports	5-4
Sample PrimeTime Timing Report	5-5
Running PrimeTime	5-6

Conclusion	5-6
------------------	-----

Section III. Power Estimation & Analysis

Revision History	Section III-1
------------------------	---------------

Chapter 6. Early Power Estimation

Introduction	6-1
Excel-Based Power Calculator	6-1
Estimating Power in the Design Cycle	6-3
Quartus II Power Report File	6-6
Conclusion	6-8
References	6-8

Chapter 7. Simulation-Based Power Estimation

Introduction	7-1
Power Estimation in the Quartus II Software	7-2
Estimating Power with EDA Simulation Tools	7-4
Scripting Support	7-7
Simulation-Based Power Estimation Settings	7-7
Generate a Power Input File	7-8
Conclusion	7-8
References	7-8

Section IV. On-Chip Debugging

Revision History	Section IV-2
------------------------	--------------

Chapter 8. Quick Design Debugging Using SignalProbe

Introduction	8-1
Using SignalProbe	8-1
Reserving SignalProbe pins	8-2
Adding SignalProbe Sources	8-3
Assigning I/O Standards	8-4
Adding Registers for Pipelining	8-4
Performing a SignalProbe Compilation	8-5
Running SignalProbe with Smart Compilation	8-7
Understanding SignalProbe Routing Failures	8-7
Understanding the Results of a SignalProbe Compilation	8-8
Scripting Support	8-9
Reserving SignalProbe Pins	8-10
Adding SignalProbe Sources	8-10
Assigning I/O Standards	8-10
Adding Registers for Pipelining	8-11
Run SignalProbe Automatically	8-11
Run SignalProbe Manually	8-11

Enable or Disable All SignalProbe Routing	8–11
Running SignalProbe with Smart Compilation	8–12
Allow SignalProbe to Modify Fitting Results	8–12
Conclusion	8–12

Chapter 9. Design Debugging Using the SignalTap II Embedded Logic Analyzer

Introduction	9–1
Including the SignalTap II Logic Analyzer in Your Design	9–2
Using the STP File to Create an Embedded Logic Analyzer	9–3
Using the MegaWizard Plug-In Manager to Create your Embedded Logic Analyzer	9–8
Programming the Device for SignalTap II Analysis	9–12
View Data Samples	9–12
Advanced Features	9–12
Preserving FPGA Memory	9–13
Creating Complex Triggers	9–14
Using External Triggers	9–17
Embedding Multiple Analyzers in One FPGA	9–20
Faster Compilations	9–20
Time Bars and Next Transition	9–22
Saving Captured Data	9–22
Converting Captured Data to Other File Formats	9–22
Creating Mnemonics for Bit Patterns	9–23
Buffer Acquisition	9–23
Capturing Data to a Specific RAM Type	9–24
FPGA Resources Used by SignalTap II	9–24
Using SignalTap II in a Lab Environment	9–25
Remote Debugging Using SignalTap II	9–25
Signal Preservation	9–28
Tappable Signals	9–29
Timing Preservation with SignalTap II Logic Analyzer	9–29
Using SignalTap II Logic Analyzer to Simultaneously Debug Multiple Designs	9–29
Locating a Node in the Chip Editor	9–31
Design Example: Preserving Timing	9–32
Design Example: Using SignalTap II Logic Analyzers in SOPC Builder Systems	9–35
Conclusion	9–35

Chapter 10. Design Analysis and Engineering Change Management with Chip Editor

Introduction	10–1
Background	10–1
Using the Chip Editor in Your Design Flow	10–2
Chip Editor Overview	10–3
Chip Editor Floorplan	10–4
Bird's Eye View	10–5
First (Highest) Level View	10–6
Second Level View	10–7
Third Level View	10–8
Resource Property Editor	10–9

The Logic Element (LE)	10-9
The Adaptive Logic Module (ALM)	10-10
Supported Changes for an LE/ALM	10-11
Properties of the Logic Element	10-12
Mode of Operation	10-12
LUT Equation	10-12
LUT Mask	10-13
Synchronous Mode	10-14
Register Cascade Mode	10-14
Properties of an ALM	10-14
LUT Mask	10-14
Extended LUT Mode	10-15
Shared Arithmetic Mode	10-15
FPGA I/O Elements	10-15
Stratix, Stratix GX, and Stratix II I/O Elements	10-15
Cyclone I/O Elements	10-17
MAX II I/Os	10-17
Supported Changes for an I/O Element	10-18
Editable Properties of I/O Elements	10-19
Modifying the PLL Using the Chip Editor	10-21
Properties of the PLL	10-21
Adjusting the Duty Cycle	10-22
Adjusting the Phase Shift	10-22
Adjusting the Output Clock Frequency	10-22
Adjusting the Spread Spectrum	10-23
Change Manager	10-23
Common Applications	10-24
Gate-Level Register Retiming	10-24
Routing an Internal Signal to an Output Pin	10-26
Adjust the Phase Shift of a PLL to Meet I/O Timing	10-27
Correcting a Design Flaw	10-27
Example Design: Meeting I/O Timing	10-27
Running the Quartus II Timing Analyzer	10-33
Generating a Netlist for Other EDA Tools	10-33
Generating a Programming File	10-33
Conclusion	10-34

Chapter 11. In-System Updating of Memory & Constants

Overview	11-1
Device & Megafunction Support	11-2
Creating In-System Configurable Memory and Constants	11-3
Running the In-System Memory Content Editor	11-4
Instance Manager	11-5
Making Changes	11-6

Viewing Memory & Constants in the Hex Editor	11-7
Programming the Device Using the In-System Memory Content Editor	11-8
Conclusion	11-9

Section V. Formal Verification

Revision History	Section V-1
------------------------	-------------

Chapter 12. Cadence Incisive Conformal Support

Introduction	12-1
Formal Verification	12-1
Equivalence Checking	12-1
Generating the VO File & Incisive Conformal Script	12-2
Comparing Designs Using Incisive Conformal Software	12-8
Black Boxes in the Incisive Conformal Flow	12-8
Running the Incisive Conformal Software	12-9
Known Issues & Limitations	12-11
Conclusion	12-11

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: *June 2004*
Part number: *qii53001-2.0*
- Chapter 2. Synopsys VCS Support
Revised: *June 2004*
Part number: *qii53002-2.0*
- Chapter 3. Cadence NC-Sim Support
Revised: *August 2004*
Part number: *qii53003-2.0*
- Chapter 4. Quartus II Timing Analysis
Revised: *June 2004*
Part number: *qii53004-2.0*
- Chapter 5. Synopsys PrimeTime Support
Revised: *June 2004*
Part number: *qii53005-2.0*
- Chapter 6. Early Power Estimation
Revised: *June 2004*
Part number: *qii53006-2.0*
- Chapter 7. Simulation-Based Power Estimation
Revised: *June 2004*
Part number: *qii53007-2.0*
- Chapter 8. Quick Design Debugging Using SignalProbe
Revised: *June 2004*
Part number: *qii53008-2.0*
- Chapter 9. Design Debugging Using the SignalTap II Embedded Logic Analyzer
Revised: *June 2004*
Part number: *qii53009-2.0*

Chapter 10. Design Analysis and Engineering Change Management with Chip Editor

Revised: *June 2004*
Part number: *qii53010-2.0*

Chapter 11. In-System Updating of Memory & Constants

Revised: *August 2004*
Part number: *qii53012-1.0*

Chapter 12. Cadence Incisive Conformal Support

Revised: *June 2004*
Part number: *qii53011-2.0*



About this Handbook

This handbook provides comprehensive information about the Altera® Quartus® II design software, version 4.0.

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	lit_req@altera.com (1)	lit_req@altera.com (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} , lqdesigns 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>.pdf file.</p>
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	<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.	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.

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 Chapters 1 to 3.

Chapter(s)	Date / Version	Changes Made
1	June 2004 v2.0	<ul style="list-style-type: none"> • Updates to tables, figures. • New functionality for Quartus 4.1.
	Feb. 2004 v1.0	Initial release
2	June 2004 v2.0	<ul style="list-style-type: none"> • Updates to tables, figures. • New functionality for Quartus 4.1.
	Feb. 2004 v1.0	Initial release
3	Aug. 2004 v2.1	<ul style="list-style-type: none"> • New functionality for Quartus 4.1 SP1
	June 2004 v2.0	<ul style="list-style-type: none"> • Updates to tables, figures. • New functionality for Quartus 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 RTL and gate-level timing simulations for either VHDL or Verilog HDL designs targeting 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 simulation or a gate-level timing simulation.

This document describes ModelSim-Altera software version 5.8c and the ModelSim PE software version.



This document contains references to features available in the Altera Quartus® II software version 4.1. Please visit the Altera web site, available at www.altera.com/quartus for information on this Quartus II software version.

Background

The ModelSim-Altera software version 5.8c 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 simulations 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
100% VHDL, Verilog, mixed-HDL support	option	option	Supports only single-HDL simulation
Complete HDL debugging environment	✓	✓	✓
Optimized direct compile architecture	✓	✓	✓
Industry-standard scripting	✓	✓	✓
Flexible licensing	✓	option	✓

Table 1–1. Comparison of ModelSim Versions (Part 2 of 2)

Verilog PLI (1) support. Interfaces Verilog designs to customer C code and third-party software	✓	✓	✓
VHDL FLI (2) 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 (3) 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 specific ModelSim-Altera software version is compatible with the specific Quartus II software version. 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 1–20.

Table 1–2. Compatibility Between Software Versions

ModelSim-Altera Software	Quartus II Software (1)
ModelSim-Altera software version 5.7c	Quartus II software version 3.0
ModelSim-Altera software version 5.7e	Quartus II software version 4.0
ModelSim-Altera software version 5.8c	Quartus II software version 4.1

Note to Table 1–2:

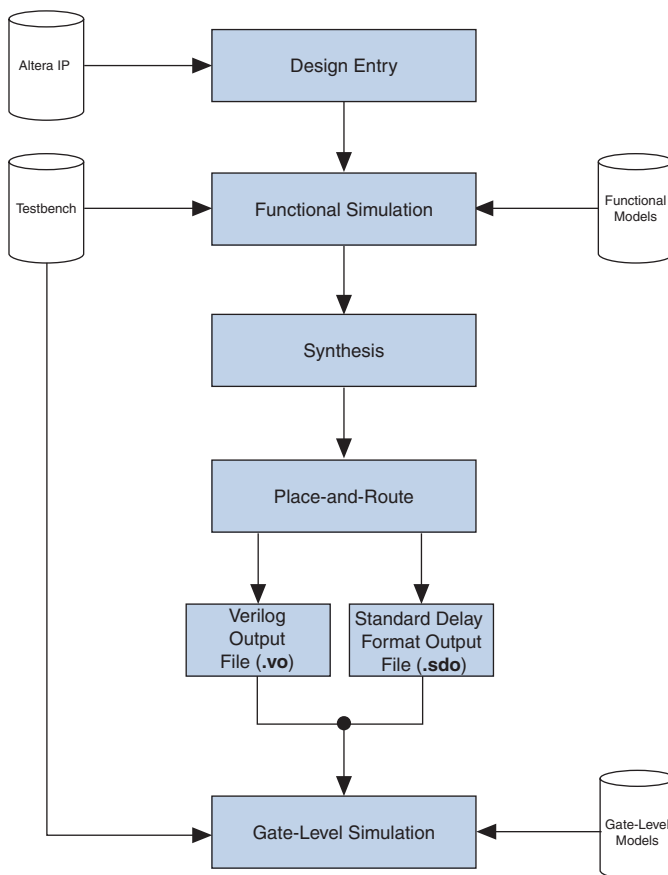
(1) ModelSim-Altera precompiled libraries are updated with Quartus II release and service packs and are generally available for download on Altera’s web site.

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
- Gate-level timing simulations

Figure 1-1. Altera Design Flow with ModelSim-Altera and Quartus II Software



Functional RTL Simulation

Functional RTL simulations verify the functionality of the design before synthesis and 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.

Gate-Level Timing Simulation

Place-and-route in the Quartus II software produces a design netlist (**.vo** or **.vho** file) and a Standard Delay Format (SDF) output (**.sdo**) file used for a gate-level timing simulation in the ModelSim-Altera software. The design netlist output file is a netlist of the design mapped to architecture-specific primitives such as logic elements and I/O elements. The SDF file contains delay information for each architecture primitive and routing element specific to the design. Together, these files provide an accurate simulation of the design for the selected Altera FPGA architecture.

Functional RTL Simulation

A functional RTL simulation is performed before a gate-level simulation and verifies the functionality of the design before 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 and Altera Megafunction Functional RTL 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 **220model_87.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 1–3 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\sim_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)

Note 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.

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	<Quartus II installation directory>\eda\sim_lib\ (1)
ModelSim-Altera (PC)	<ModelSim-Altera installation directory>\altera\<HDL>\altera_mf\
ModelSim-Altera (UNIX)	<ModelSim-Altera installation directory>/modeltech/altera/<HDL>/altera_mf/

Note to Table 1–4:

- (1) For Model Technology's ModelSim, use the files provided with the Quartus II software.

Simulating VHDL Designs

The following instructions will help you to perform a functional RTL simulation for VHDL designs in the ModelSim-Altera software.



The following steps assume you have already created a ModelSim project.

Create Simulation Libraries



Creating a simulation library is not required if you are using the ModelSim-Altera software.

Simulation libraries are needed to 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.

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** box.
4. Click **OK**.

```
vlib altera_mf↵  
vmap altera_mf altera_mf↵  
  
vlib lpm↵  
vmap lpm lpm↵
```



The name of the libraries should be `altera_mf` (for Altera megafunctions) and `lpm` (for LPM and Megawizard-generated entities).

Compile Simulation Models into Simulation Libraries



The following steps are not required for the ModelSim-Altera software.

1. Choose **Add to Project** (File menu) and select **Existing File**.
2. Browse to the `<quartus installation folder>/eda/sim_lib` and add the necessary simulation model files to your project.
3. Select the simulation model file and select **Properties** (View menu).
4. Set the **Compile to Library** to the correct library.



The `altera_mf.vhd` should be compiled into the `altera_mf` library. The `220model.vhd` should be compiled into the `lpm` library.

```
vcom -work altera_mf <quartus installation  
directory>/eda/sim_lib/altera_mf_components.vhd> ↵
```

```
vcom -work altera_mf <quartus installation folder>/eda/sim_lib  
/altera_mf.vhd> ↵
```

```
vcom -work lpm <quartus installation folder/eda/sim_lib /220pack.vhd> ↵
```

```
vcom -work lpm <quartus installation folder/eda/sim_lib /220model.vhd> ↵
```

Compile Testbench and Design Files into Work Library

1. Select **Compile All** (Compile menu) or click the **Compile All** toolbar icon
2. Resolve compile-time errors before proceeding to “Loading the Design” below.

```
vcom -work work <my_testbench.vhd> <my_design_files.vhd>↵
```

Loading the Design

1. Select **Simulate** (Simulate menu).
2. Expand the work library in the **Simulate** dialog box.
3. Select the top-level design unit (your testbench). Select **OK** in the **Simulate** dialog box.

```
vsim work.<my_testbench>↵
```

Running the Simulation

1. Choose **Signals and Wave** (View menu).

```
view signals↵
view wave↵
```

2. Drag signals to monitor from the **Signals** window and drop them into the **Wave** window.

```
add wave /<signal name>↵
```

3. At the prompt type the following:

```
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-Altera software.



The following steps assume you have already created a ModelSim project.

Create Simulation Libraries



Creating a simulation library is not required for the ModelSim-Altera software.

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.

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** box.
4. Click **OK**.

```
vlib altera_mf
vmap altera_mf altera_mf
```

```
vlib lpm
vmap lpm lpm
```



The name of the libraries should be `altera_mf` (for Altera megafunctions) and `lpm` (for LPM and Megawizard-generated entities).



This process is not required for the ModelSim-Altera version because a set of pre-compiled libraries are created when you install the ModelSim-Altera software.

Compile Simulation Models into Simulation Libraries



The following steps are not required for the ModelSim-Altera software.

1. Choose **Add to Project** (File menu) and select **Existing File**.
2. Browse to the `<quartus installation folder>/eda/sim_lib` and add the necessary simulation model files to your project.
3. Select the simulation model file and select **Properties** (View menu).
4. Set the **Compile to Library** to the correct library.



The `altera_mf.v` should be compiled into the `altera_mf` library. Compile the **220model.v** into the `lpm` library.

```
vlog -work altera_mf <quartus installation folder/eda/sim_lib  
/altera_mf.v> ↵
```

```
vlog -work lpm <quartus installation folder/eda/sim_lib /220model.v>↵
```

Compile Testbench and Design Files into Work Library

1. Select **Compile All** (Compile menu) or click the **Compile All** toolbar icon
2. Resolve compile-time errors before proceeding to “Loading the Design” below.

```
vlog -work work <my_testbench.v> <my_design_files.v>↵
```

Loading the Design

1. Select **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 1–5](#) 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). Select **OK** in the **Simulate** dialog box.

```
vsim -L <location of the altera_mf library> -L <location of the lpm  
library> work.<my_testbench>↵
```

Running the Simulation

1. Choose **Signals and Wave** (View menu).

```
view signals↵  
view wave↵
```

2. Drag signals to monitor from the **Signals** window and drop them into the **Wave** window.

```
add wave /<signal name>↵
```

3. At the prompt type the following:

```
run <time period>↵
```

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 power-up data via a hexadecimal (.hex) or Memory Initialization File (.mif). The LPM_FILE parameter included in the MegaWizard-generated file 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 through the Quartus II software.

Neither ModelSim software version can directly read a HEX file or MIF format. Therefore, to allow functional simulation of Altera memory blocks in the ModelSim software, you must perform the following steps:

1. Convert a HEX file or MIF to a RAM Initialization File (.rif).
2. Modify of the MegaWizard-generated file.
3. Compile the **nopli.v** file.

Converting a HEX File or MIF to a RIF

A RIF is an ASCII text file that you use with tools from electronic design automation (EDA) vendors. Create a RIF by converting an existing MIF or HEX file using the **Export** command in the Quartus II software. This option is available through the **File** menu.

Modifying the MegaWizard-Generated File

You must modify the MegaWizard-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 *s<path to Quartus II installation>\eda\sim_lib* directory. This file contains the following definition:

```
`define NO_PLI 1
```

This basic definition instructs the ModelSim compile to read in the RIF.

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 **.vo** for Verilog HDL and **.vho** for VHDL output files. 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 output files, perform the following steps:

1. Choose **EDA Tool Settings** (Assignments menu).
2. In the **Simulation** Tool box:
 - a. If you are using ModelSim-Altera, select **ModelSim OEM (VHDL/Verilog HDL output from Quartus II)**.
 - b. If you are using Model Technology's ModelSim, select **ModelSim (VHDL/Verilog HDL output from Quartus II)**.
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 various ModelSim-Altera precompiled device libraries.

Table 1–5. Various ModelSim-Altera Precompiled Device Libraries	
Library	Description
maxii	Precompiled library for MAX [®] II devices
stratixii	Precompiled library for Stratix [®] II devices
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)
cyclone	Precompiled library for Cyclone [™] 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 devices and ARM [®] -based Excalibur [™] 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
max	Precompiled library for MAX 7000 and MAX 3000 device designs

Table 1–6 shows the location of the timing simulation libraries in the ModelSim-Altera software for Verilog HDL for PCs.

Table 1–6. Location of Timing Simulation Libraries for ModelSim-Altera for Verilog HDL on a PC (Part 1 of 2)	
Library	Verilog HDL
maxii	<ModelSim-Altera installation directory>\altera\verilog\maxii\
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\

Table 1–6. Location of Timing Simulation Libraries for ModelSim-Altera for Verilog HDL on a PC (Part 2 of 2)

Library	Verilog HDL
cyclone	<ModelSim-Altera installation directory>\altera\verilog\cyclone\
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\
max	<ModelSim-Altera installation directory>\altera\verilog\max\

Table 1–7 shows the location of the timing simulation libraries in the ModelSim-Altera software for VHDL for PCs.

Table 1–7. Location of Timing Simulation Library Files for ModelSim-Altera for VHDL on a PC

Library	VHDL
maxii	<ModelSim-Altera installation directory>\altera\vhd\maxii\
stratixii	<ModelSim-Altera installation directory>\altera\vhd\stratixii\
stratix	<ModelSim-Altera installation directory>\altera\vhd\stratix\
stratixgx	<ModelSim-Altera installation directory>\altera\vhd\stratixgx\
stratixgx_gxb	<ModelSim-Altera installation directory>\altera\vhd\stratixgx_gxb\
cyclone	<ModelSim-Altera installation directory>\altera\vhd\cyclone\
apexii	<ModelSim-Altera installation directory>\altera\vhd\apexii\
apex20ke	<ModelSim-Altera installation directory>\altera\vhd\apex20ke\
apex20k	<ModelSim-Altera installation directory>\altera\vhd\apex20k\
flex10ke	<ModelSim-Altera installation directory>\altera\vhd\flex10ke\
flex6000	<ModelSim-Altera installation directory>\altera\vhd\flex6000\
mercury	<ModelSim-Altera installation directory>\altera\vhd\mercury\
max	<ModelSim-Altera installation directory>\altera\vhd\max\

Table 1–8 shows the location of the timing simulation libraries in the ModelSim-Altera software for Verilog HDL for UNIX.

Table 1–8. Location of Timing Simulation Libraries for ModelSim-Altera for Verilog HDL with UNIX	
Library	Verilog HDL
maxii	<ModelSim-Altera installation directory>/modeltech/altera/verilog/maxii/
stratixii	<ModelSim-Altera installation directory>/modeltech/altera/verilog/stratixii/
stratix	<ModelSim-Altera installation directory>/modeltech/altera/verilog/stratix/
stratixgx	<ModelSim-Altera installation directory>/modeltech/altera/verilog/stratixgx/
stratixgx_gxb	<ModelSim-Altera installation directory>/modeltech/altera/verilog/stratixgx_gxb/
cyclone	<ModelSim-Altera installation directory>/modeltech/altera/verilog/cyclone/
apexii	<ModelSim-Altera installation directory>/modeltech/altera/verilog/apexii/
apex20k	<ModelSim-Altera installation directory>/modeltech/altera/verilog/apex20k/
apex20ke	<ModelSim-Altera installation directory>/modeltech/altera/verilog/apex20ke/
mercury	<ModelSim-Altera installation directory>/modeltech/altera/verilog/mercury/
flex10ke	<ModelSim-Altera installation directory>/modeltech/altera/verilog/flex10ke/
flex6000	<ModelSim-Altera installation directory>/modeltech/altera/verilog/flex6000/
max	<ModelSim-Altera installation directory>/modeltech/altera/verilog/max/

Table 1–9 shows the location of the timing simulation libraries in the ModelSim-Altera software for VHDL for UNIX.

Table 1–9. Location of Timing Simulation Libraries for ModelSim-Altera for VHDL with UNIX (Part 1 of 2)	
Library	VHDL
maxii	<ModelSim-Altera installation directory>/modeltech/altera/vhdl/maxii/
stratixii	<ModelSim-Altera installation directory>/modeltech/altera/vhdl/stratixii/
stratix	<ModelSim-Altera installation directory>/modeltech/altera/vhdl/stratix/
stratixgx	<ModelSim-Altera installation directory>/modeltech/altera/vhdl/stratixgx/
stratixgx_gxb	<ModelSim-Altera installation directory>/modeltech/altera/vhdl/stratixgx_gxb/
cyclone	<ModelSim-Altera installation directory>/modeltech/altera/vhdl/cyclone/
apexii	<ModelSim-Altera installation directory>/modeltech/altera/vhdl/apexii/
apex20k	<ModelSim-Altera installation directory>/modeltech/altera/vhdl/apex20k/
apex20ke	<ModelSim-Altera installation directory>/modeltech/altera/vhdl/apex20ke/
mercury	<ModelSim-Altera installation directory>/modeltech/altera/vhdl/mercury/
flex10ke	<ModelSim-Altera installation directory>/modeltech/altera/vhdl/flex10ke/

Table 1–9. Location of Timing Simulation Libraries for ModelSim-Altera for VHDL with UNIX (Part 2 of 2)

Library	VHDL
flex6000	<ModelSim-Altera installation directory>/modeltech/altera/vhdl/flex6000/
max	<ModelSim-Altera installation directory>/modeltech/altera/vhdl/max/

If you are using the ModelSim-Modeltech version for your timing simulation, libraries are available in the Quartus II software at the following location: <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.

1. Select **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**.

```
vlib stratixii
vmap stratixii stratixii
```

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.

1. Select **Add to Project** (File menu), then select **Existing File**.
2. Browse to the *<quartus installation folder>/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. Set the **Compile to Library** to the correct library.

```
vcom -work altera_mf <quartus installation folder>/eda/sim_lib  
/stratixii_components.vhd> ↵
```

```
vcom -work altera_mf <quartus installation folder>/eda/sim_lib  
/stratixii.vhd> ↵
```

Compile Testbench and VHO into Work Library

1. Choose **Compile All** (Compile menu) or click the **Compile All** toolbar icon.
2. Resolve any compile time errors before proceeding to *Loading the Design*.

```
vcom -work work <my_testbench.vhd> <my_vhdl_output_file.vho> ↵
```

Loading the Design

1. Select **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 select **OK** in the **Simulate** dialog box.

```
vsim -sdftyp work.<my_testbench> ↵
```

Running the Simulation

1. Choose **Signals and Wave** (View menu).

```
view signals↵
view wave↵
```

2. Drag signals to monitor from the **Signals** window and drop them into the **Wave** window.

```
add wave /<signal name>↵
```

3. At the prompt type the following:

```
run <time period>↵
```

Simulation 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



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.

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**.

```
vlib stratixii↵
vmap stratixii stratixii↵
```

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.

1. Select **Add to Project** (File menu) and select **Existing File**.
2. Browse to the `<quartus installation folder>/eda/sim_lib` and add the necessary simulation model files to your project.
3. Select the simulation model file and select **Properties** (View menu).
4. Set the **Compile to Library** to the correct library.

```
vlog -work stratixii <quartus installation folder>/eda/sim_lib
/startixii_atoms.v> ↵
```

Compile Testbench and VO into Work Library

1. Select **Compile All** (Compile menu) or click the **Compile All** toolbar icon.
2. Resolve any compile time errors before proceeding to *Loading the Design*.

```
vlog -work work <my_testbench.v> <my_verilog_output_file.vo> ↵
```

Loading the Design

1. Select **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 select **OK** in the **Simulate** dialog box.

```
vsim -L <location of the gate level simulation library> -
work.<my_testbench> ↵
```

Running the Simulation

1. Choose **Signals and Wave** (View menu).


```
view signals↵
view wave↵
```

2. Drag signals to monitor from the **Signals** window and drop them into the **Wave** window.

```
add wave /<signal name>↵
```

3. At the prompt type the following:

```
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. Select **EDA Tool Settings** (Assignments menu) and set the **Simulation Tool Name** to one of the following:

ModelSim (Verilog Output from Quartus II)
 ModelSim (VHDL Output from Quartus II)
 ModelSim-Altera (Verilog Output from Quartus II)
 ModelSim-Altera (VHDL Output from Quartus II)



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.

UNIX workstations only: to run ModelSim automatically from the Quartus II software using the NativeLink feature, you must add the following environment variables to your .cshrc:

```
QUARTUS_INI_PATH <ModelSim installation directory> ↵
```

Software Licensing & Licensing Set-Up

License the ModelSim-Altera software through a parallel port software guard (T-guard), FIXEDPC 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.



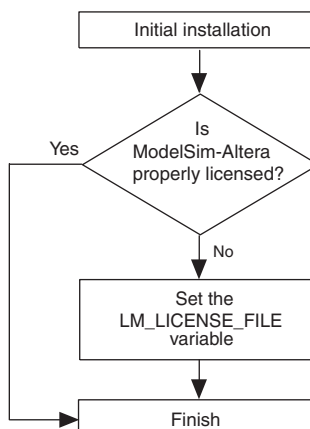
USB is not supported.

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 and timing simulation on their designs. Proper verification of designs at the functional 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 a getting-started guide to using the Synopsys VCS software to simulate designs targeting Altera® FPGAs. It provides a step-by-step explanation of how to perform functional 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 4.1. For more information on the Quartus II software version 4.1, go to the Altera web site at www.altera.com.

Software Requirements

In order to properly simulate your design using VCS, you must first install the Quartus II software.

Table 2–1 shows the supported Quartus II-VCS version compatibility.

<i>Table 2–1. Supported Quartus II & VCS Software Version Compatibility</i>	
Synopsys	Altera
VCS software version 7.0	Quartus II software version 3.0
VCS software version 7.0.1	Quartus II software version 4.0
VCS software version 7.1.1	Quartus II software version 4.1



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.

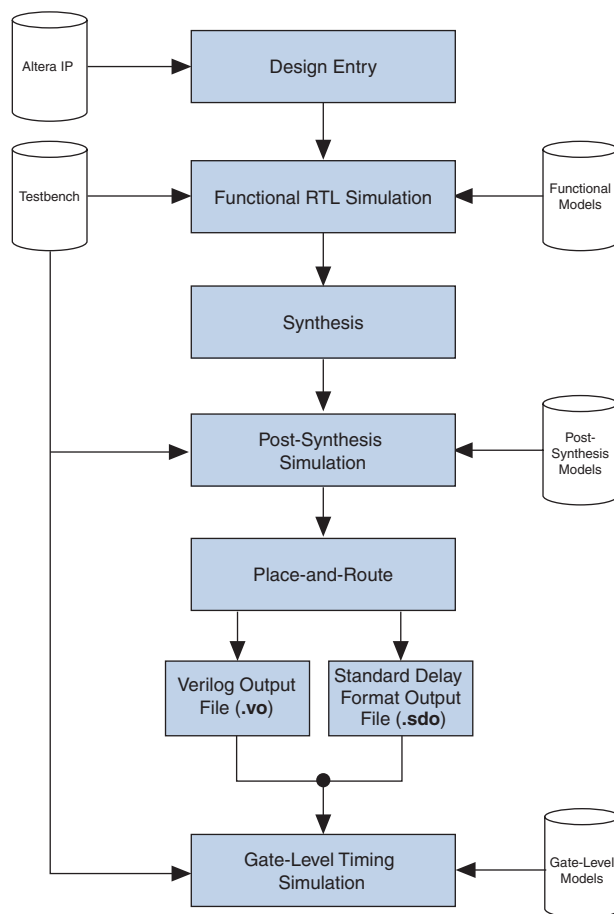
Using VCS in the Quartus II Design Flow

The VCS software supports the following types of simulation:

- 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 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 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 library of parameterized modules (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 StratixGX design you will need to compile the stratixgx_mf.v, sgate.v, & 220model.v

The files in Table 2–2 are installed with a Quartus II installation. You can find these files in the `<path to Quartus II installation>\eda\sim_lib` directory.

The following VCS command describes the command-line syntax to perform a functional 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 MegaWizard-generated file 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 through the Quartus II software.

However, the VCS software cannot read a HEX file or MIF 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 MegaWizard-generated file
3. Compile the **nopli.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 through the **File** menu while the Quartus II memory editor is open.

Modifying the MegaWizard-Generated File

You must modify the MegaWizard-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 basic 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:

```
vcs -R <path to Quartus installation>\eda  
    \sim_lib\nopli.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:

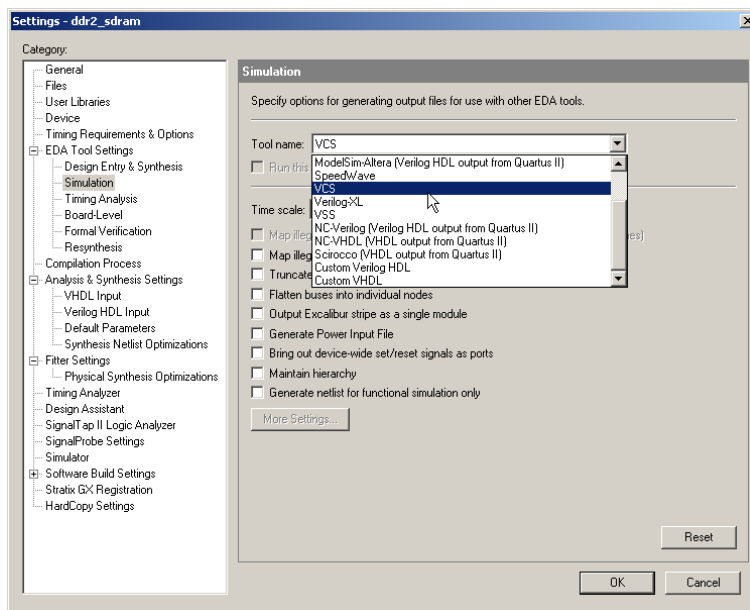
1. Perform Analysis & Synthesis:

Choose **Start > Start Analysis & Synthesis** (Processing menu).

2. Enable the **Generate Netlist for Functional Simulation Only**:

Choose **Settings** (Assignments menu). In the **Category** list, select **EDA Tool Settings** (expand if necessary) > **Simulation**. In the Simulation section of the window, choose **VCS** in the **Tool name** list, as shown in [Figure 2-2](#).

Figure 2-2. Setting the Tool Name to VCS in the Settings Window



3. Run the EDA Netlist Writer:

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 2–7](#), can be used to perform a post-synthesis simulation in VCS.

The following VCS commands describes the command-line syntax used to perform a post-synthesis simulation with the appropriate device family library listed in [Table 2–3 on page 2–7](#):

```
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 in Quartus II

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 compilation has completed successfully, set the **Tool name** to **VCS**:

Choose **Settings** (Assignments menu). In the **Category** list, select **EDA Tool Settings** (expand if necessary) > **Simulation**. In the Simulation section of the window, choose **VCS** in the **Tool name** list, as shown in [Figure 2–2](#).

3. Run the EDA Netlist Writer:

Choose **Start > Start EDA Netlist Writer** (Processing menu).

During the EDA Netlist Writer stage, the Quartus II software produces a Verilog output (.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 will be located in the *<project folder>/simulation/VCS* directory. These files, along with the device family library listed in [Table 2-3](#), can be used to perform a gate-level timing simulation in VCS.

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 timing netlist.vo> -v <altera device family library.v>
```

Table 2-3. Altera Gate-Level Simulation Libraries

Library Files	Description
apex20k_atoms.v	Atom libraries for APEX™ 20K designs
apex20ke_atoms.v	Atom libraries for APEX 20KE, APEX 20KC, and Excalibur™ designs
apexii_atoms.v	Atom libraries for APEX II designs
cyclone_atoms.v	Atom libraries for Cyclone™ designs
flex6000_atoms.v	Atom libraries for FLEX® 6000 designs
flex10ke_atoms.v	Atom libraries for FLEX 10KE and ACEX® 1K designs
max_atoms.v	Atom libraries for MAX® 3000 and MAX 7000 designs
mercury_atoms.v	Atom libraries for Mercury™ designs
stratix_atoms.v	Atom libraries for Stratix designs
stratixgx_atoms.v stratixgx_hssi_atoms.v	Atom libraries for Stratix GX designs
stratixii_atoms.v	Atom libraries for Stratix II designs
maxii_atoms.v	Atom libraries for MAX II designs
cycloneii_atoms.v	Atom libraries for Cyclone II designs
hc_stratix_atoms.v	Atom libraries for HardCopy Stratix designs

Transport Delays

VCS filters out all pulses that are shorter than the propagation delay between elements. Enabling the transport delay switches in VCS prevents the simulation tool from filtering out these pulses. Use the following switches to ensure that all signal pulses are seen in the simulation results:

+transport_path_delays

Use this switch 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 compile-time options.

+transport_int_delays

Use this switch 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 compile-time options.



For more information on either of these switches refer to the *VCS User Guide* installed with the tool.

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 Compile Switches

The VCS software has a set of switches that help you simulate your design. [Table 2–4](#) lists some of the switches that are available.

<i>Table 2–4. Device Family Library Files</i>	
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 alteramf.v). The VCS software looks in this file for module definitions that are found in the source code.

Table 2–4. Device Family Library Files

Library	Description
<code>-y <library directory></code>	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.
<code>+compsdf</code>	Indicates that the VCS compiler includes the back-annotated SDF file in the compilation.
<code>+cli</code>	After successful completion of compilation, Command Line Interface (CLI) Mode is entered.
<code>+race</code>	Specifies that the VCS software generate a report that indicates all of the race conditions in the design. Default report name is race.out .
<code>-P</code>	Compiles user-defined Programming Language Interface (PLI) table files.
<code>-q</code>	Indicates the VCS software runs in quiet mode. All messages are suppressed.



For more information on any VCS switch, 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 invoked by using the `-I` compile-time switch 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 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 switch. 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` switches, 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 `vcuser.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

Run procedures and create settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For more information about Tcl scripting, 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 it:

```
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 easily and accurately perform simulations, post-synthesis simulations, gate-level functional and timing simulations.

Introduction

This chapter is a getting started guide to using the Cadence NC family of simulators in Altera® FPGA design flows. The NC family is comprised of the 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 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 4.1 software. See the Altera web site at www.altera.com for information on the Quartus II version 4.1 software.

Software Requirements

You must first install the Quartus II software before using it with Cadence NC simulators. The Quartus II/Cadence interface is automatically installed when the Quartus II software is installed on your computer.

Table 3–1 shows which Cadence NC simulator version is compatible with a specific Quartus II software version.

Table 3–1. Compatibility between Software Versions			
Cadence NC Simulators (UNIX)	Cadence NC Simulators (PC)	Cadence NC Simulators (Linux)	Quartus II Software
Version 5.0 s005 Version 5.1 s012	Version 5.0 s006 Version 5.1 s010	Version 5.0 p001 Version 5.0 p001	Version 4.0 Version 4.1



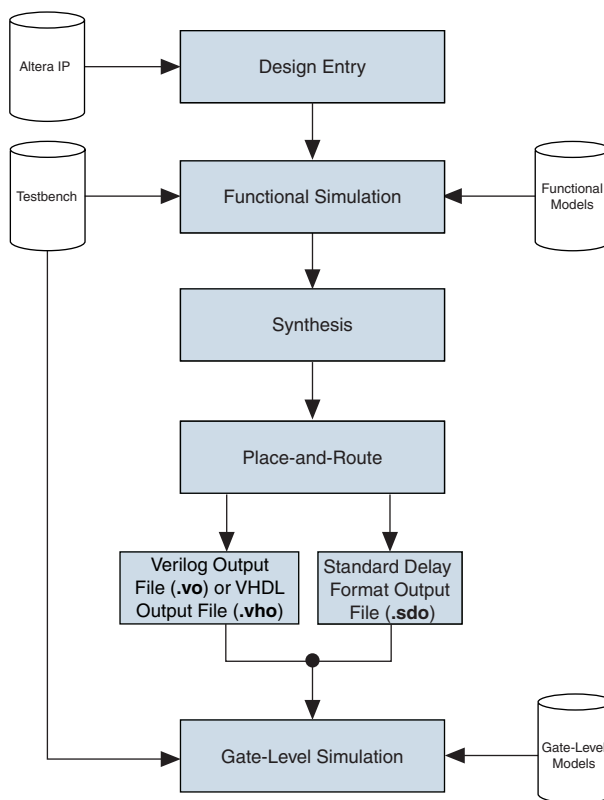
See the *Quartus II Installation & Licensing for PCs* or *Quartus II Installation & Licensing for UNIX and Linux Workstations* manuals for more information on installing the software, and the directories that are created during the Quartus II installation.

Simulation Flow Overview

The Cadence NC software supports the following simulation flows:

- Functional/RTL simulation
- Gate-level timing simulation

Figure 3–1 shows the Quartus II/Cadence design flow.

Figure 3–1. Altera Design Flow with Cadence NC Simulators

Functional/RTL Simulation

Functional/RTL simulation verifies the functionality of your design. When you perform a functional simulation with Cadence NC 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 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.

Gate-Level Timing Simulation

After performing place-and-route in the Quartus II software, the software generates a Verilog Output File (.vo) or VHDL Output File (.vho) and a Standard Delay Format (SDF) Output File (.sdo) for gate-level timing simulation. The netlist files map your design to architecture-specific primitives. The SDO 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 3–18](#) for more information on how to perform this simulation.

Operation Modes

You can use either the command-line mode or graphical user interface (GUI) mode to simulate your design with NC simulators. To simulate in command-line mode, use the files shown in [Table 3–2](#).

You can launch the NC GUI in UNIX or PC environments by typing `nclaunch`  at a command prompt.



This chapter describes how to perform simulation using both the command-line and the GUI.

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/NC Simulation Flow Overview

The Quartus II/Cadence NC simulation flow is described below. A more detailed set of instructions are given in “[Functional/RTL Simulation](#)” on page 3–5 and “[Gate-Level Timing Simulation](#)” on page 3–18.

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, e.g., 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 using the GUI. 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, the design hierarchy must be defined in a process called elaboration. Use **ncelab** in command-line mode or choose **Elaborator** (Tools menu) in GUI mode 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 Cadence NC tools.

Set Up Your Environment


This section describes how to set up your working environment for the Quartus II/NC-Verilog or NC-VHDL software interface.



(For UNIX workstations only) 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. (For UNIX workstations only) Add the following environment variables to your **.cshrc** file:

```
setenv QUARTUS_ROOTDIR /usr/quartus
setenv CDS_INST_DIR <NC installation directory>
```

2. Add **\$CDS_INST_DIR/tools/bin** directory to the **PATH** environment variable in your **.cshrc** file. Make sure this path are 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.

Following is an example setting these environment variables.

Setting 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 NC simulators, you must set up libraries using a file named **cds.lib**. The **cds.lib** file is an ASCII text file that maps logical library names—e.g., your working directory or the location of resource libraries such as models for LPM functions—to their physical directory paths. When you launch an NC tool, the tool reads **cds.lib** to determine which libraries are accessible and where they are located. NC tools include 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 (e.g., technology or cell libraries) and a user **cds.lib** file. The following sections describe how to create/edit a **cds.lib** file, including:

- Basic Library Setup
- LPM Function & Altera Megafunction Libraries

Basic Library Setup

You can create **cds.lib** 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** for Verilog HDL contains the lines:

```
DEFINE lib_std /usr1/libs/std_lib
DEFINE worklib ../worklib
```

Using Multiple **cds.lib** Files

Use the `INCLUDE` or `SOFTINCLUDE` statements 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, you must use an `INCLUDE` or `SOFTINCLUDE` statement in the **cds.lib** file to include your default **cds.lib** in addition to the `DEFINE` statements. The syntax 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 another **cds.lib** files such as **cdsvhdl.lib** and **cdsvlog.lib**. These files contain library definitions for IEEE libraries, Synopsys libraries, etc.

Create cds.lib: Command-Line Mode

To edit **cds.lib** from the command line, perform the following steps:

1. Create a directory for the work library and any other libraries you need using the command:

```
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 cds.lib: GUI Mode

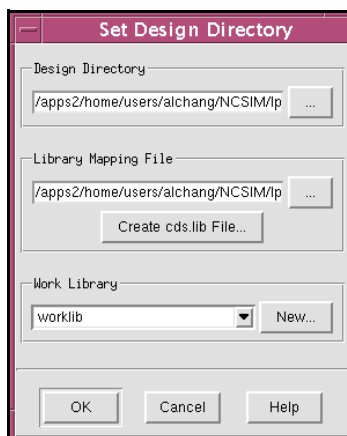
To create **cds.lib** using the GUI, perform the following steps:

1. Type `nclaunch` at the command line to launch the GUI.
2. If the **NCLaunch** window is not in multiple step mode, switch to multistep mode by selecting **Switch to Multiple Step** (File menu).
3. Change your design directory by selecting **Set Design Directory** (File menu).

The **Set Design Directory** window opens, as shown in [Figure 3–2](#).

4. Click on the Browse button (. . .) to navigate to your project directory.
5. Click **Create cds.lib File** and choose the appropriate libraries to be included in the **New cds.lib File** dialog box.
6. Click **New** under **Work Library**.
7. Enter your new work library name, e.g., **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 **cds.lib** by right-clicking the **cds.lib** filename in the right pane and choosing **Edit** from the 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 NC tools.



Many LPM functions and Altera megafunctions use memory files. You must convert the memory files for use with NC tools before simulating. To convert these files into a format the NC tools can read follow the instruction in section “[Simulating a Design with Memory](#)” on page 3–10.

Altera provides megafunction behavioral descriptions in the files shown in [Table 3–1](#). 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 (3)	stratixgx_mf.v(4)	stratixgx_mf.vhd(4) stratixgx_mf_components.vhd(4)
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) As an alternative you can map to the precompiled library `<Quartus II installation>/eda/sim_lib/modelsim/<verilog|vhd|>/altgxb`
- (4) The ATGXB 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, NC tools cannot read a HEX or MIF. Perform the following steps to convert these files into a format the tools can read.

1. Convert your HEX or MIF into a RAM Initialization File (**.rif**) by performing the following steps in the Quartus II software:



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 to view their usage.

- a. Open the HEX or MIF file.
 - b. Choose **Export** (File menu).
 - c. If necessary, in the **Export** dialog box, select a target directory in the **Save in** list.
 - d. Select a file to overwrite in the **Files** list or type the file name in the **File name** box.
 - e. If necessary, in the **Save as type** list, select **RAM Initialization File (.rif)**.
 - f. Click **Export**.
2. Using a text editor, modify the `lpm_file` parameter in the megafunction's wizard-generated file to point to the RIF. Alternatively, you can rerun the wizard and point to the RIF as the memory initialization file.

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"
```

Compile Source Code & Testbenches

When using NC simulators, you compile files with **ncvlog** (for Verilog HDL files or **ncvhdl** (for VHDL files). Both **ncvlog** and **ncvhdl** perform syntax checks 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 library directory.

Compilation: 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 also need to compile the Altera-provided functional models. The following commands shows examples of each.

Verilog HDL:

```
ncvlog -WORK lpm 220model.v ↵
ncvlog -WORK altera_mf altera_mf.v ↵
```

If your design also 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 ↵
```

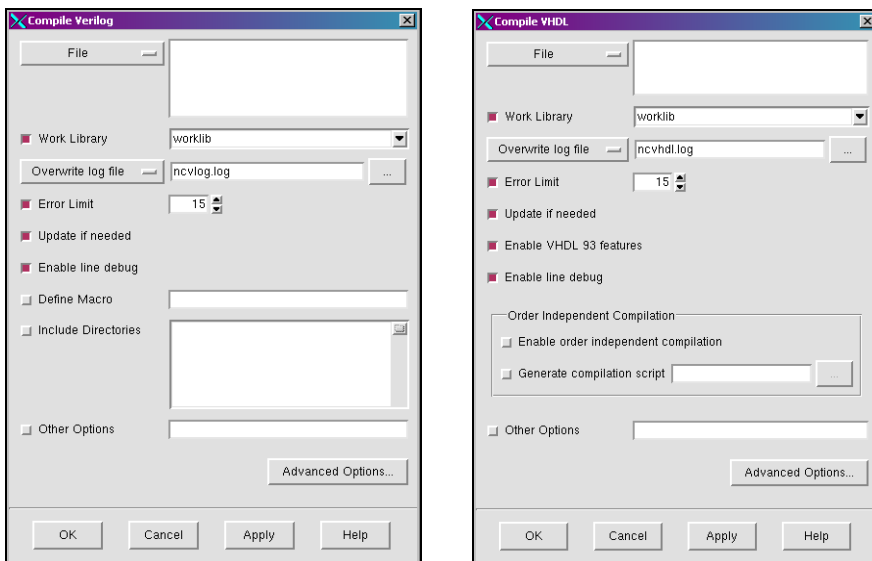
Compilation: GUI Mode

To compile using the 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 displays 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. With NC simulators, 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: 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 ↵
```



In verilog, if a timescale has been specified, the **TIMESCALE** option is not necessary.

You can set your simulation timescale using the `-TIMESCALE <arguments>` option. For example:

```
ncelab -TIMESCALE 1ps/1ps ↵
worklib.lpm_ram_dp_test:entity ↵
```



To view the elements in your library and which views are available, use the **ncls** program. For example the command `ncls -library worklib` ↵ displays all of the cells and their views in your current worklib directory.



For more information on the **ncls** program, see the Cadence NC-Verilog Simulator Help or Cadence NC-VHDL Simulator Help.



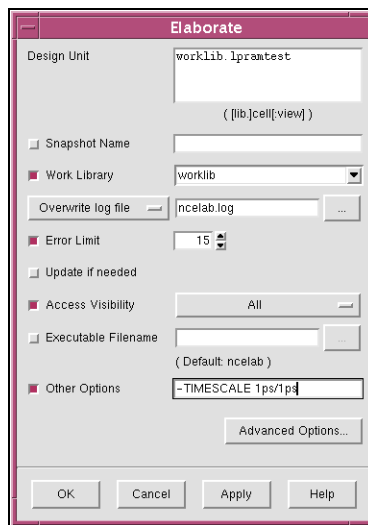
If you are running the NC-Verilog simulator using multistep invocation, run **ncelab** with command-line options as shown above. You can specify the arguments in any order, but parameters to options must immediately follow the options they modify.

Elaboration: GUI Mode

To elaborate using the GUI, perform the following steps.

1. Expand your current working library in the right panel.
2. Select and open the entity/module name you want to elaborate.
3. Right-click the view you want to display.
4. Choose **NCElab**. The **Elaborate** dialog box opens. Or you can choose **Elaborator** from the Tools menu.
5. Set the simulation timescale using the command `-TIMESCALE <arguments>` under **Other Options**. See [Figure 3-4](#).

Figure 3-4. Elaborating the Design



6. Click **OK** in the **Elaborate** dialog box to begin elaboration. The dialog box closes and returns you to **NCLaunch**.

Add Signals to View

You use a 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 NC-Sim user manual.

Adding Signals: Command-Line Mode

To create an SHM database you 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 *Cadence documentation* for details.

Table 3–4. SHM Database System Tasks

System Task	Description
<code>\$shm_open ("<filename>.shm") ;</code>	Open database. You can provide a filename; if you do not specify one, the default is waves.shm . You must create a database before you can open it; if one does not exist, the tools create it 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>	Save the database.
<code>\$shm_close;</code>	Close the database.

Following shows a simple example.

Example SHM Verilog HDL Code

```
initial
begin
    $shm_open ("waves.shm");
    $shm_probe ("AS");
end
```

For more information on these system tasks, see the NC-Sim user manual.

Adding Signals: GUI Mode

To add signals in GUI mode, perform the following steps.

1. Load the design.
 - a. Click the + icon next to the **Snapshots** directory to expand it.
 - b. Right-click the **lib.cell:view** you want to simulate, and choose **NC-Sim** (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 as shown in [Figure 3–5](#) and [Figure 3–6](#).

Figure 3–5. SimVision Console

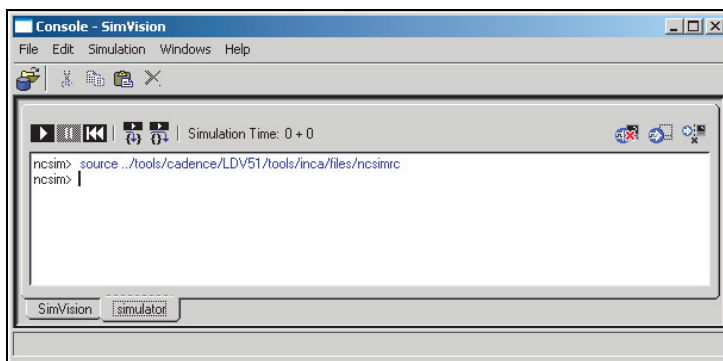
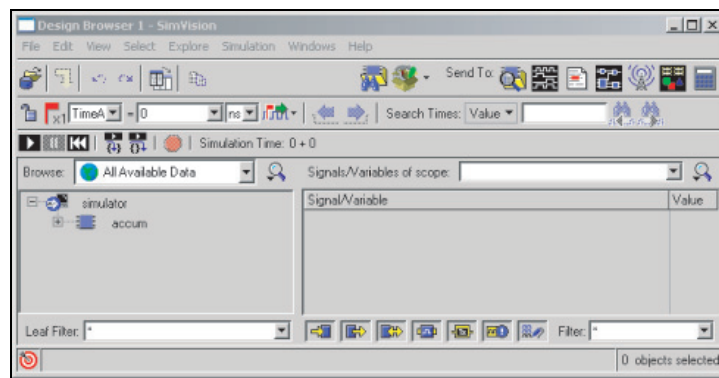


Figure 3–6. SimVision Design Browser



2. In the **Design Browser** window, select a module in the left panel and select the signals you want to view in the waveform by selecting the signal names in the **Signals/Variable** list.
3. To send the selected signals to the Waveform Viewer:

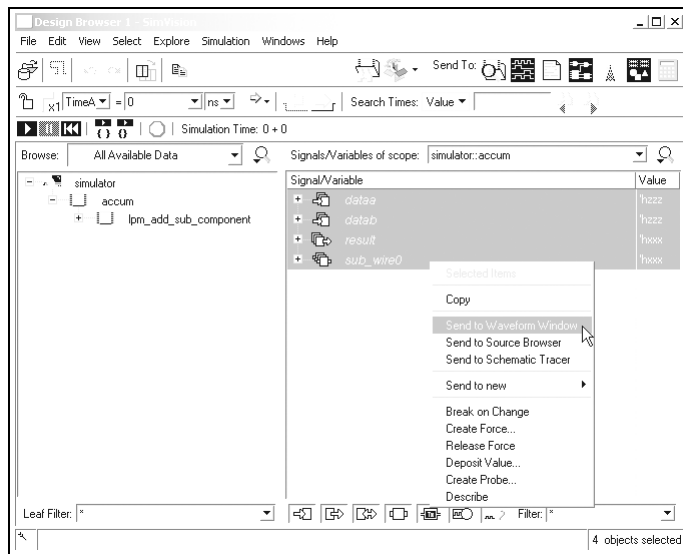
Select the **Send to Waveform Viewer** icon in the **Send To** area (the upper-right area of the **Design Browser** window),

or

Choose the **Send to Waveform Window** item in the right mouse click menu, as shown in [Figure 3–7](#).

A waveform window appears with all of your signals and you are now ready to simulate your testbench/design.

Figure 3–7. Selecting Signals in the Design Browser Window



Simulate Your Design

After you have compiled and elaborated your design, you can simulate using **ncsim**. The **ncsim** program loads the **ncelab**-generated snapshot 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: Command-Line Mode

To perform functional/RTL simulation of your Verilog HDL or VHDL design from the command line, use 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 -gui is used).

Functional/RTL Simulation: GUI Mode

You can run and step through simulation of your Verilog HDL or VHDL design in the GUI. Select **Run** from the **Simulation** menu to begin simulation.



If you skipped “Add Signals to View” on page 3-15, you must load the design before simulating. See step 1 “Load the design.” on page 3-16 for instructions.

Gate-Level Timing Simulation

The following sections provide detailed instructions for performing timing simulation using Quartus II output files and simulation libraries and Cadence NC tools.

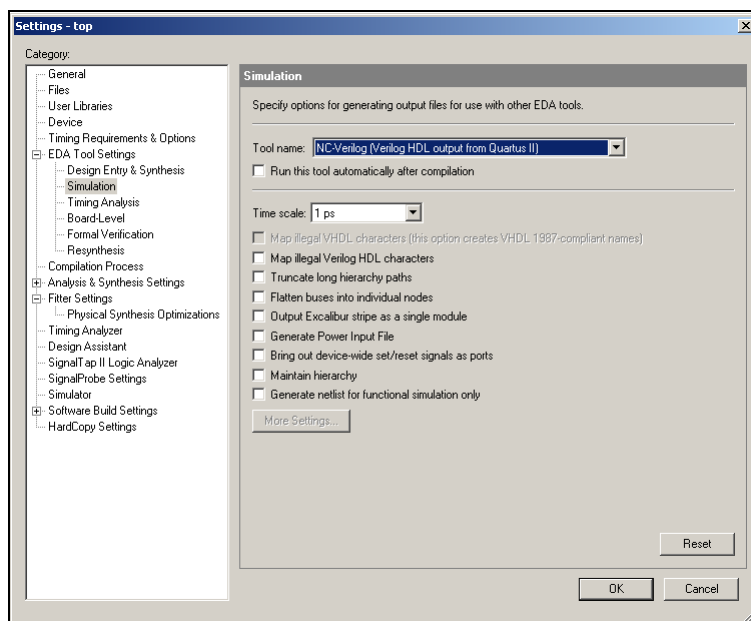
Quartus II Simulation Output Files

When you compile your Quartus II design, the software generates VO or VHO files and a 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. Click on the “plus” (+) to the left of **EDA Tool Settings** in the **Category** list. This will expand the **EDA Tool Settings** branch to show the settings.
3. Choose the **Simulation** setting. The **Simulation** page appears as shown in [Figure 3–8](#).
4. In the **Simulation** page, select **NC-Verilog (Verilog HDL output from Quartus II)** or **NC-VHDL (VHDL output from Quartus II)** in the **Tool name** list. See [Figure 3–8](#).
5. Click **OK**.
6. 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.

Figure 3–8. Quartus II EDA Tool Settings



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™ 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**. See [“Create Libraries” on page 3–20](#) 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 instructions in [“Set Up Your Environment” on page 3–5](#) for details.

Create Libraries

Create the following libraries for your simulation:

- A working 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 contains the `altgxb` megafunction, map to the precompiled Stratix GX timing simulation model libraries using the mapping *<Quartus II installation>/eda/sim_lib/ncsim/<verilog|vhd>/stratixgx_gxb* or create a new library `altgxb` using the following files in the *<Quartus II installation>/eda/sim_lib* directory:

```
stratixgx_hssi_atoms.v  
stratixgx_hssi_atoms.vhd  
stratixgx_hssi_components.vhd
```

The `altgxb` 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 3–6 and “LPM Function & Altera Megafunction Libraries” on page 3–8 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 including the following files:

- Testbench file
- Your 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 3–11 for instructions on compiling.

Elaborate the Design

When you elaborate your design, you must include the SDO file. For Verilog HDL, this process happens automatically. The Quartus II generated Verilog HDL netlist file reads the SDF file using the system task call `$sdf_annotate`. When NC-Verilog elaborates the netlist, **ncelab** recognizes the system task and automatically calls **nsdfc**. However, the `$sdf_annotate` system task call does not specify the path. Therefore, you must copy the SDO file from the Quartus II-created simulation directory to the NC working directory in which you run the **ncelab** program. After you update the path, you can elaborate the design. See “Elaborate Your Design” on page 3–13 for step-by-step instructions on elaboration.

For VHDL, the Quartus II-generated VHDL netlist file has no 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): Command Line” and “Compiling the Standard Delay Output File (VHDL Only): GUI” on page 3–22 for information on compiling the SDO file.

Compiling the Standard Delay Output File (VHDL Only): Command Line

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 a *<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>
```

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, execute the following command to elaborate the design:

```
ncelab worklib.<project name>:entity -SDF_CMD_FILE <SDF Command File> ↵
```

Compiling the Standard Delay Output File (VHDL Only): GUI

To annotate the SDO timing data in the GUI, perform the following steps:

1. Choose **SDF Compiler** (Tools menu).
2. In the **SDF File** box, specify the name of the SDO file for the project.
3. Click **OK**.

When you have finished compiling the SDO file, you can elaborate the design. See [“Elaboration: GUI Mode” on page 3–14](#) for step-by-step instructions; however, before clicking **OK** to begin elaboration, perform the following additional steps to create the SDF command file:

1. Click **Advanced Options** in the **Elaborate** dialog box.
2. Click **Annotation** in the left pane.
3. Turn on the **Use SDF File** option in the right pane.
4. Click **Edit**.
5. Browse to the location of the SDF Command File Name.
6. Browse to the location of the SDO file in the Compiled SDF File Box and click **OK**.
7. Click **OK** to save and exit the **SDF Command File** dialog box.

Add Signals to View

If you want to add signals to view, see the steps in [“Add Signals to View” on page 3–15](#).

Simulate Your Design

Simulate your design using the **ncsim** program as described in [“Simulate Your Design” on page 3–17](#).

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-x1* directory:

- **convert_hex2ver.c**
- **veriusers.c**
- **convert_hex2ver.obj**

Dynamically Link

To create a PLI dynamic library (**.so/.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, select the **Dynamic Libraries Only** option.
5. Click **Next**.
6. In the **Select Components** page, turn on the **PLI 1.0 Applications** option, 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 **veriusers.c** file that is provided with the Quartus II software.

The **veriusers.c** file is located in the following location:

<Quartus II installation>/eda/cadence/verilog-x1

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.

An example of a C compiler would be **gcc**. To allow the **PLIWIZ** to find your C compiler, ensure your path variable is set correctly.
14. Click **Next**.
15. Click **Finish**.
16. When you are 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/.dll** file, and loads them when needed.



You must modify **LD_LIBRARY_PATH** or **PATH** to include the directory location of your **.so/.dll** file.

Dynamically Load

To create a PLI library to be loaded with NC-Sim, perform the following steps:

1. Modify the **veriusers.c** file located in the following directory:

<Quartus II installation>/eda/cadence/verilog-x1

The following two examples are sections of the original and modified **veriusers.c** file.

Original veriusers.c packaged with the Quartus II software

```
s_tfcell veriusertfs[] =
{
    /*** Template for an entry:
    { usertask|userfunction, data,
      checktf(), sizetf(), calltf(), mistcf(),
      "$tfname", forwref?, Vtool?, ErrMsg? },
    Example:
    {usertask, 0, my_check, 0, my_func, my_mistcf, "$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 veriusert.c for dynamic loading

```

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);
}

```

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. In the **Select Simulator/Dynamic Libraries** page, select the **Dynamic Libraries Only** option.
5. Click **Next**.
6. In the **Select Components** page, turn on the **PLI 1.0 Applications** option, select **loadpli1**.
7. Click **Next**.
8. Type in a name into the **Bootstrap Function(s)** box.

For example, type in `my_bootstrap` into the **Bootstrap Function(s)** box.

9. Type in a name into the **Dynamic Library** box.

The name entered will be the name of your generated dynamic library.

For example, type in `convert_dyn_lib` into the **Dynamic Library** box to generate a dynamic library named **convert_dyn_lib.so**.

10. In the **PLI 1.0 Application** page, click **browse** under **PLI Source**. Files to locate the **convert_hex2ver.c** file and the modified **veriusers.c** file.

11. Click **Next**.

12. In the **Select Compiler** page, choose your C compiler from the **Select Compiler** list box.

An example of a C compiler would be **gcc**. To allow the **PLIWIZ** to find your C compiler, ensure your **Path** variable is set correctly.

13. Click **Next**.

14. Click **Finish**.

15. When asked if you want to build your targets now, click **Yes**.

Compilation generates your dynamic library, **cmd_file.nc** and **cmd_file.xl** into your local directory.

The **cmd_file.nc** and **cmd_file.xl** files contain command line options that should be used with your newly generated dynamic library file.

Use the **cmd_file.nc** command file with **ncelab** to perform your simulations.

```
ncelab worklib.mylpmrom -FILE cmd_file.nc ↵
```

Use the **cmd_file.xl** command file with **verilog-xl** or **ncverilog** to perform you simulations.

```
ncverilog -f cmd_file.xl
verilog -f cmd_file.xl
```

Statically Link

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**, 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 **veriusers.c** file that is provided with the Quartus II software.

The **veriusers.c** can be found in the following location:

<Quartus II installation>/eda/cadence/verilog-x1

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.

An example of a C compiler would be **gcc**. 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.

For **ncverilog** users, you can use the following command to perform your 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 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 3–18](#).

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 1ps 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 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, Quartus II Timing Analysis](#)
- [Chapter 5, Synopsys PrimeTime Support](#)

Revision History

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

Chapter(s)	Date / Version	Changes Made
4	June 2004 v2.0	<ul style="list-style-type: none"> • Updates to tables, figures. • New functionality for Quartus 4.1.
	Feb. 2004 v1.0	Initial release
5	June 2004 v2.0	No changes to document.
	Feb 2004 v1.0	Initial release

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 reports, giving you immediate access to this data.

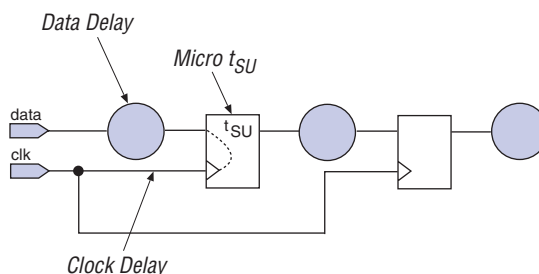
This chapter explains the basic principles of static timing analysis, and the advanced features supported by the Quartus II Timing Analyzer using TCL scripts and the Quartus II graphical user interface (GUI).

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 could result in incorrect circuit operation. This section describes the basic timing analysis measurements used 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 4-1](#) shows a diagram of clock setup time.

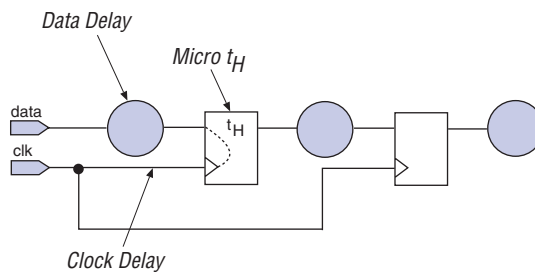
Figure 4–1. Clock Setup Time (t_{SU})

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 4–1.

$$t_{SU} = \text{Data Delay} - \text{Clock Delay} + \text{Micro } t_{SU}$$

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 4–2 shows a diagram of clock hold time.

Figure 4–2. Clock Hold Time (t_H)

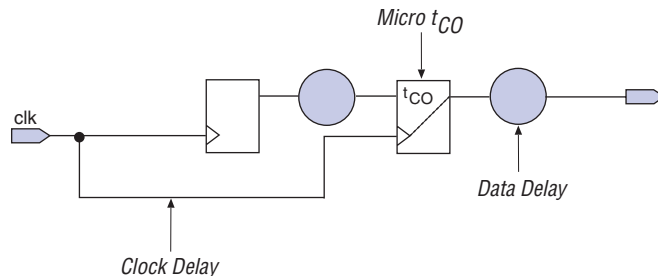
Micro t_H is the internal hold time of the register. The following equation calculates the t_H of the circuit shown in Figure 4–2.

$$t_H = \text{Clock Delay} - \text{Data Delay} + \text{Micro } t_H$$

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 4–3 shows a diagram of clock-to-output delay.

Figure 4–3. Clock-to-Output Delay (t_{CO})



The following equation calculates the t_{CO} of the circuit shown in Figure 4–3.

$$t_{CO} = \text{Clock Delay} + \text{Micro } t_{CO} + \text{Data Delay}$$

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 also make t_{PD} assignments between an input 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.

Slack

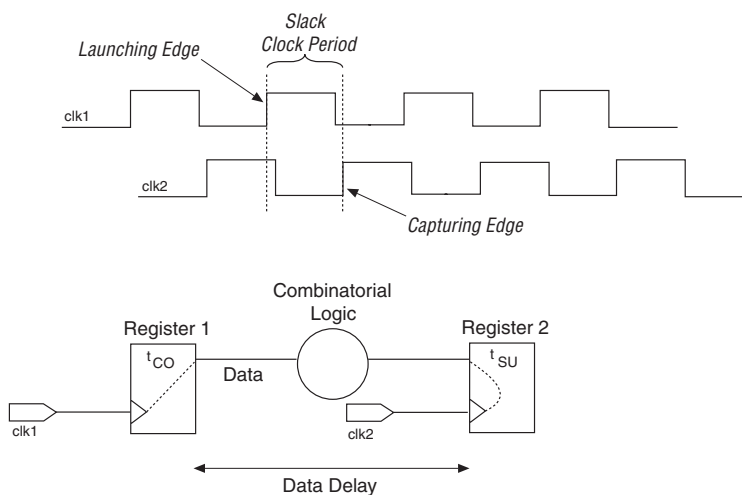
Slack is the margin by which a timing requirement (e.g., f_{MAX}) is met or not met. Positive slack indicates the margin by which a requirement is met. Negative slack indicates the margin by which a requirement was not met. The Quartus II software determines slack with the following equations.

$$\text{Slack} = \text{Required clock period} - \text{Actual clock period}$$

$$\text{Slack} = \text{Slack clock period} - (\text{Micro } t_{CO} + \text{Data Delay} + \text{Micro } t_{SU})$$

Figure 4-4 shows a slack calculation diagram.

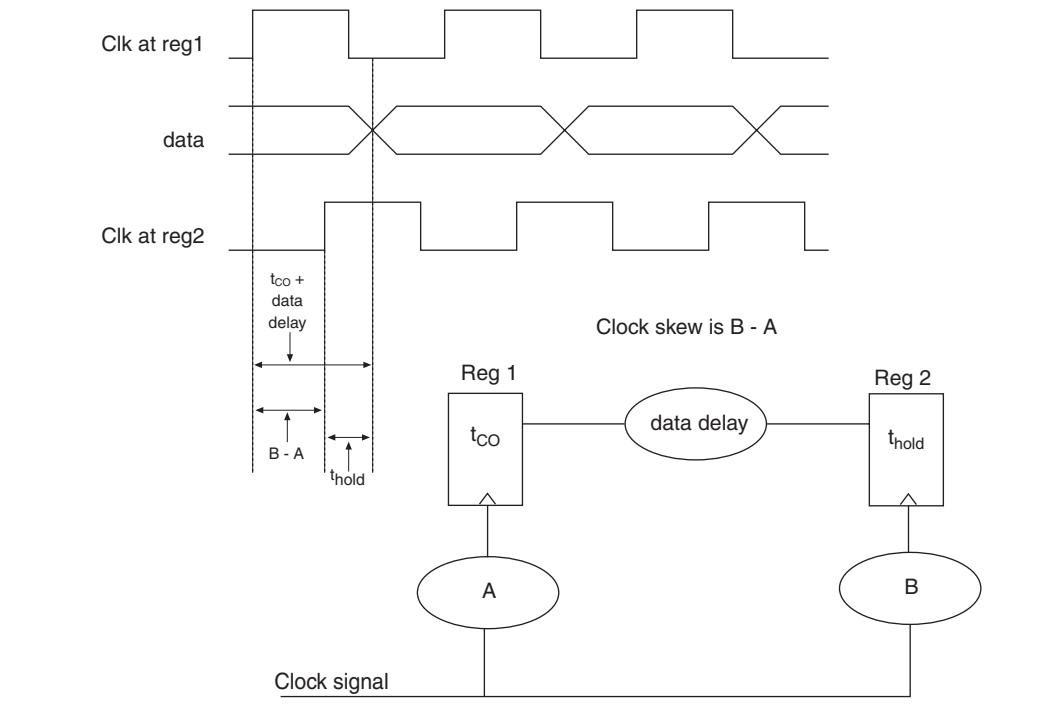
Figure 4-4. Slack Calculation Diagram



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 4-5). 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. Since the Quartus II software only reports hold time slack for paths that have hold time violations, only negative slacks are reported.

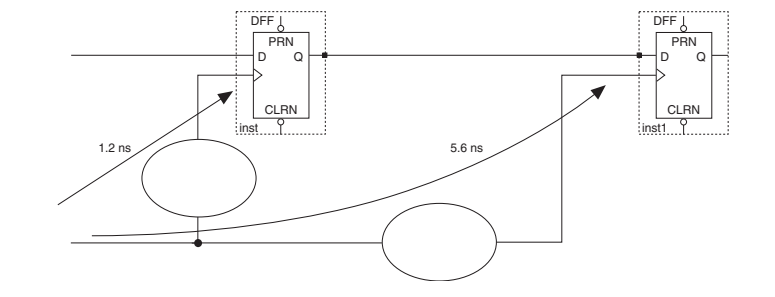
Figure 4–5. Hold Time Slack



Clock Skew

Clock skew is the difference in arrival time of a clock signal at two different registers (Figure 4–6). 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 4–6. Clock Skew



Executing Tcl Script-Based Timing Commands

You can make timing assignments, perform timing analysis, and analyze results in the Quartus II software GUI or with Tcl commands. You can use simple Tcl commands to perform customized timing reporting, and you can write scripts with advanced timing analysis commands to perform complex timing analysis and reporting.

You can use the command-based timing analyzer in an interactive shell mode where you can run timing analysis Tcl scripts.

To run the timing analyzer in interactive shell mode, type the following command:

```
quartus_tan -s
```

To run a Tcl script, type the following command:

```
quartus_tan -t <tcl file>
```

The following commands are frequently needed for executing timing-related scripts:

- `Package require ::quartus:<advanced_timing>` (Different packages are required for a different set of commands.)
- `project_open <project_name>` (Open the project in the project directory.)
- `create_timing_netlist` (Timing information is created in the memory for analysis.)
- `project_close` (This command should be executed at the end of every script.)

The remainder of this chapter includes Tcl command examples for making timing assignments and performing timing analysis. Refer to the Quartus II Command-Line and Tcl API Help for complete information about the above commands, other Tcl commands related to timing analysis and reporting, and the complete Tcl command reference.

To run the Tcl API Help, type the following command:

```
quartus_sh --qhelp
```

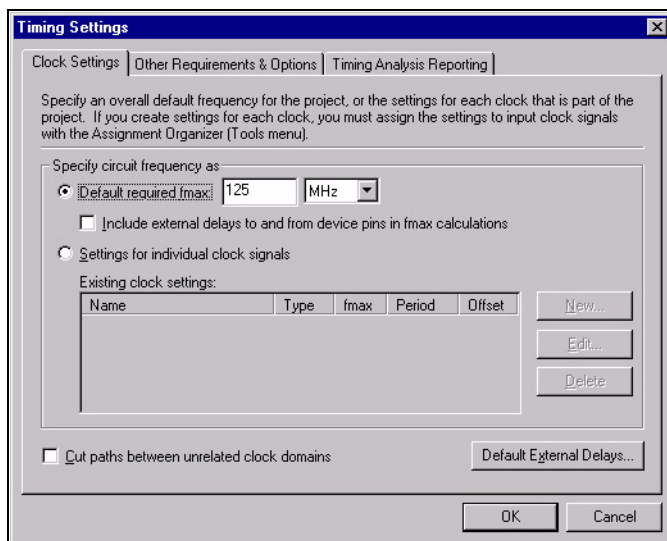
Setting up the Timing Analyzer

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 make global timing assignments in the **Timing Requirements & Options** page of the **Timing Settings** dialog box (Assignments menu), shown in [Figure 4-7](#).

Figure 4-7. Timing Settings Dialog Box



You can set global t_{SU} , t_{CO} , and t_{PD} requirements, as well as minimum t_{Hr} , 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 4-28](#).

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). Altera® recommends that you define all clock requirements and relationships in your design to ensure accurate timing analysis results.

Clocks can also be specified by executing tcl scripts.

- Usage for absolute clocks: `create_base_clock -fmax <fmax> [-duty_cycle <duty cycle>] [-target <name>] [-no_target] [-entity <entity>] [-disable] <clock_name>`
- Example for absolute clock: `create_base_clock -fmax 50ns -duty_cycle 50 clk50`
- Usage for relative clocks: `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>`
- Example for relative clock: Clk2_3 is created based on predefined clock clk10

```
create_relative_clock -base_clock -multiply 2 -divide
3 clk10 clk2_3
```

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

Use this timing assignment to assign a previously-created individual clock requirement to a pin or node in the design. The Timing Wizard makes this assignment automatically.

Input Maximum Delay

Use this timing assignment to specify the maximum allowable delay of a signal from an external 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 4–8 shows a block diagram of the input delay.

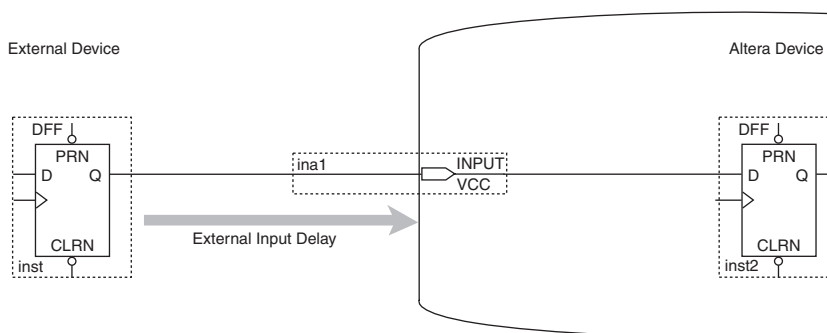
For example, input maximum delay of 2ns can be set on a predefined group called "input_pins" by using -max option. Timegroup command is used to gather signal names by using wild card into a group for timing assignment purpose as shown in the example.

```
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 during an open project are not committed to .qsf file unless the export_assignments command is explicitly executed. If a close_project command is executed, the assignments are committed into the .qsf also.

Figure 4–8. External Input Delay



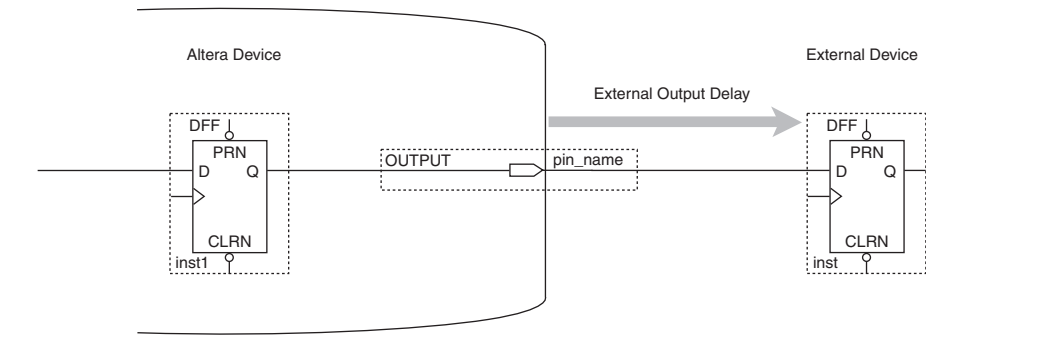
Output Maximum Delay

Use this timing assignment to specify the maximum allowable delay of a signal from the specified output pin to an external 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 4–9 shows a block diagram of the external output delay.

Script usage of output minimum delay:

```
set_output_delay [-clk_ref <clock>] -to <output_pin>
[-min] [<value>]
```

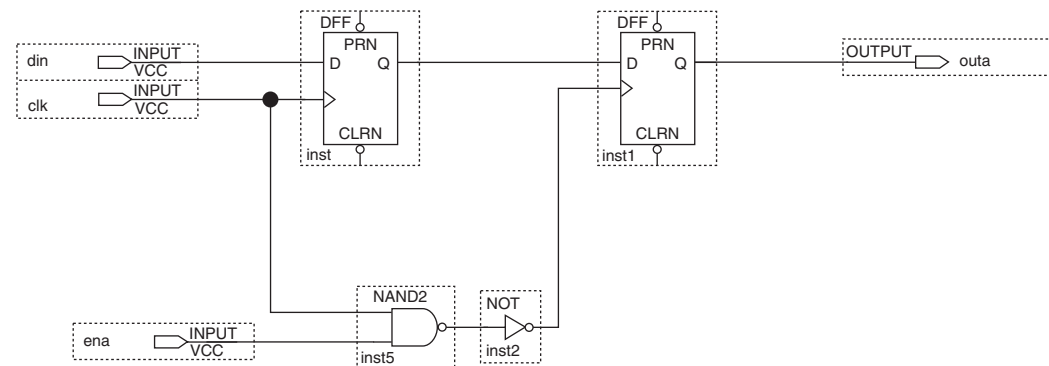
Figure 4–9. Output Delay



Inverted Clock

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. An example of a complex logic structure is shown in [Figure 4–10](#).

Figure 4–10. Complex Logic Structure

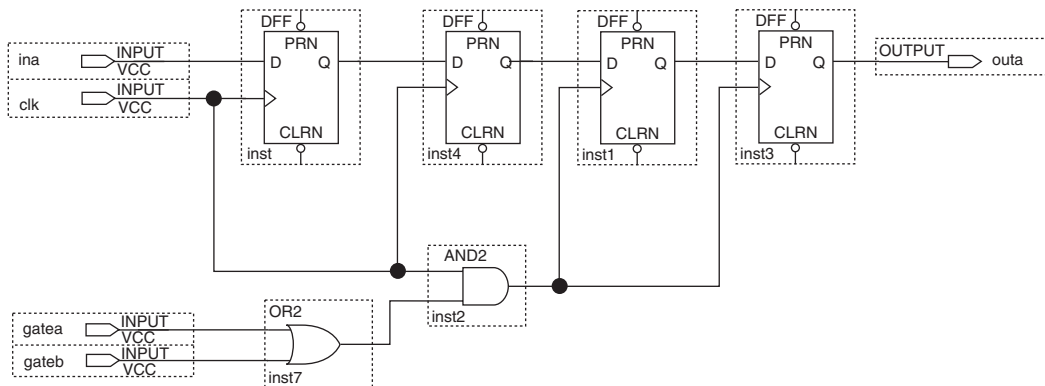


In the example shown in Figure 4-10, when the enable is active, the clock is inverted. Under these circumstances, you should make an inverted clock assignment to the register, `inst1`, to ensure that the Timing Analyzer recognizes the inverted clock.

Not a Clock

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 4-11.

Figure 4-11. Not a Clock Diagram



In Figure 4-11, 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, you can use the following Tcl command to explicitly remove a clock from timing analysis:

```
set_instance_assignment -name NOT_A_CLOCK -to clk
```

t_{CO} Requirement

Individual t_{CO} assignments have priority over global assignments. You can make t_{CO} assignments to either the pin, the output register, or from the output register to the pin.

t_H Requirement

Individual t_H assignments have priority over global assignments. You can make t_H assignments to either the pin, the input register, from the pin to the input register, or from the clock pin to the input register.

t_{PD} Requirement

Individual t_{PD} assignments have priority over global assignments. You can make t_{PD} assignments from input pins to output pins, from input pins to registers, from registers to registers, from registers to output pins, and as a single point assignment to an input pin.

t_{SU} Requirement

Individual t_{SU} assignments have priority over global assignments. You can make t_{SU} assignments to either the input pin, the input register, from the input pin to the input register, or from the clock pin to the input register.

Timing Wizard

The Timing Wizard helps you make global timing assignments. Choose **Wizards > Timing Wizard** (Assignments menu) to start it. You can use either the Timing Wizard or the **Timing Requirements & Options** page of the **Settings** dialog box to specify global timing requirements.

Timing Analysis Reporting in the Quartus II Software

The Quartus II timing analysis report is displayed as sections in the Compilation report. The timing report includes an f_{MAX} and slack for all clock pins.



If there are no timing assignments for the design, the Timing Analyzer does not generate slack reports for the 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.

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.

If a design contains individual t_{SU} , t_H or t_{CO} assignments and does not contain global t_{SU} , t_H or t_{CO} assignments, only the individual assignments are reported in the timing analysis reports. If a design contains individual t_{SU} , t_H , or t_{CO} assignments and you need a timing report for t_{SU} , t_H , or t_{CO}

on all I/O pins, you must set global t_{SU} , t_{H} , or t_{CO} assignments to generate a timing report on the pins not specified by the individual timing assignments.

Advanced Timing Analysis



The Quartus II software performs timing analysis of designs containing paths that cross clock domains and designs that contain multicyle paths. This section describes these advanced features.

For detailed instructions on how to use these or any of the Quartus II Timing Analyzer features, see the Quartus II Help.

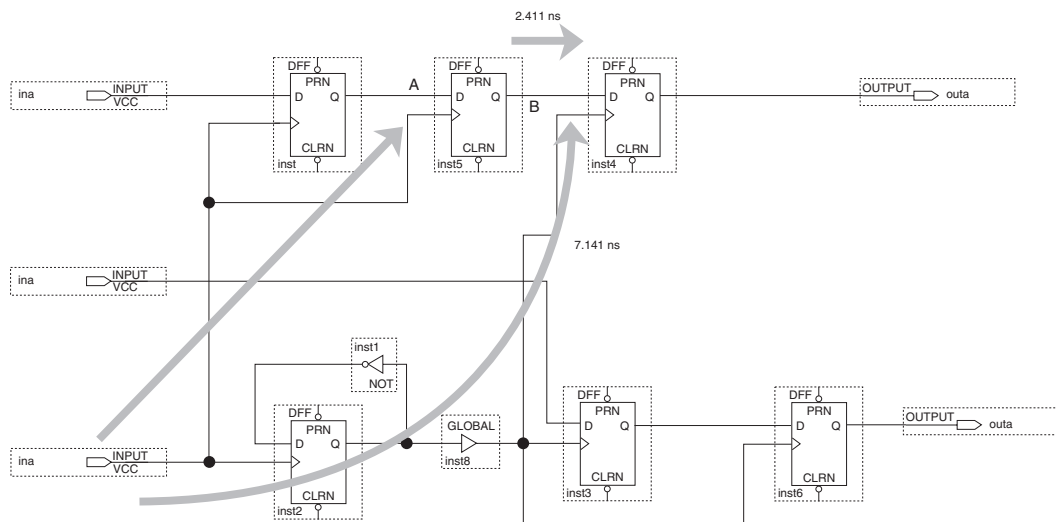
Clock Skew

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. An example of this is shown in [Figure 4–12](#).

Figure 4–12. Derived Clocks Example



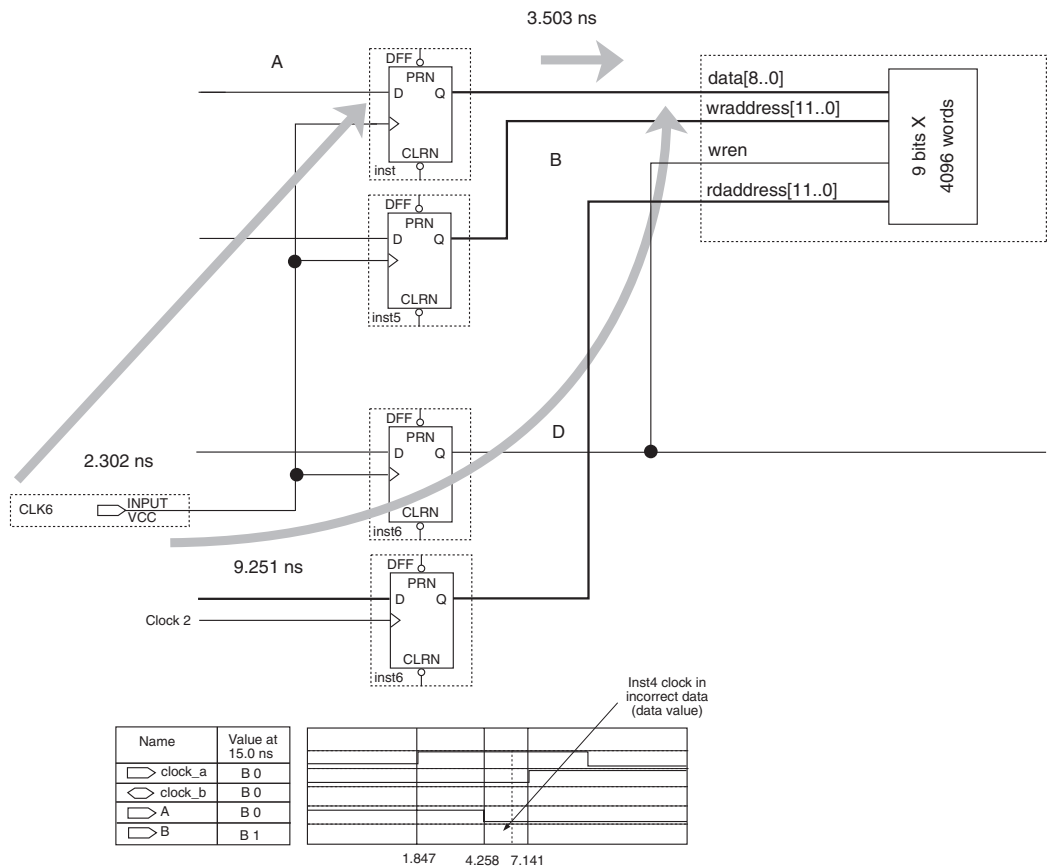
In [Figure 4–12](#), the longest clock path is 7.141 ns from `clock_a` to destination register `inst4`. The shortest clock path is 1.847 ns from `clock_a` to the source register `inst5`. This creates a clock skew of 5.294 ns.

The shortest register-to-register data path between the source and destination register is 2.411 ns. The micro hold delay of the destination register is 0.710 ns. Thus, the clock skew is longer than the data path (5.294 ns > 2.411 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 on the latch. An example is shown in [Figure 4–13](#). 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 4–13. Clock Skew



Multiple Clock Domains

Multiclock circuits are designs that have more than one clock. After you specify clock settings, the Quartus II software analyzes timing for register-to-register paths controlled by 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 4–30 for more information.

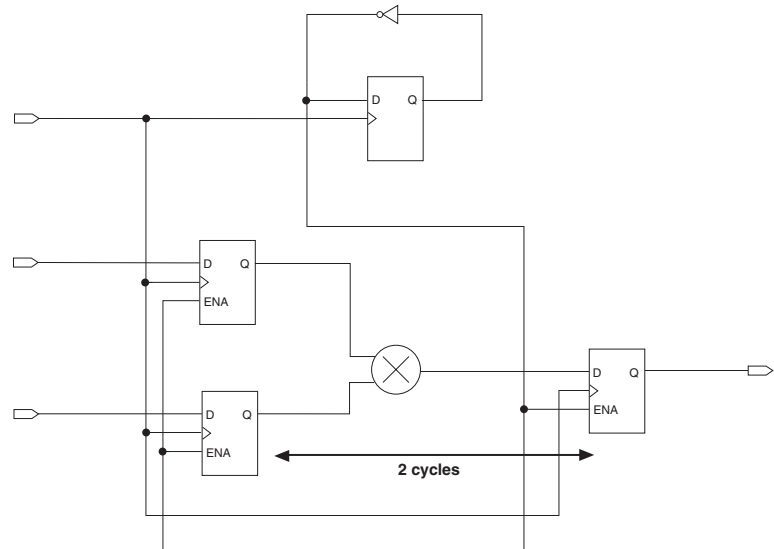
To correctly perform multiclock timing analysis, you must define the absolute clock, specify a desired f_{MAX} or clock period, and define other clocks and their relationships, if any, to the absolute clock. Then, assign these settings to the clock pins that supply the design’s clock signals. Upon successful compilation, the Quartus II Timing Analyzer automatically verifies circuit operability.

Multicycle Assignments

Multicycle paths are paths between registers that intentionally require more than one clock cycle to become stable. For example, a register may need to trigger a signal on every second or third rising clock edge.

Figure 4–14 shows an example of a design with a multicycle path between the multiplier's input registers and output register.

Figure 4–14. Example Diagram of a Multicycle Path

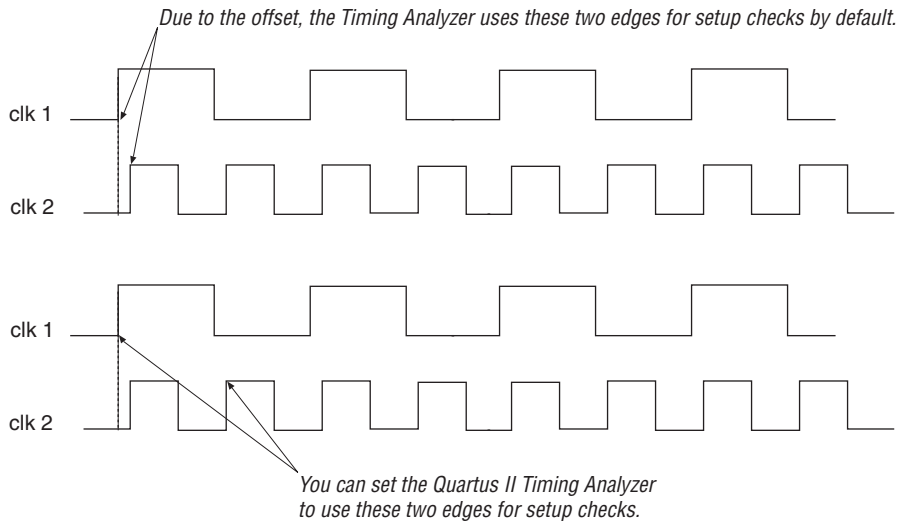


Multicycle Assignment

A **Multicycle** assignment specifies the number of clock cycles required before a register should latch a value. Multicycle assignments delay the latch edge, relaxing the required setup relationship.

Figure 4–15 shows a timing diagram for a multicycle path that exists in a design with related clocks, with a small offset between the clocks.

Figure 4–15. Multicycle Paths with Offset Between Clocks

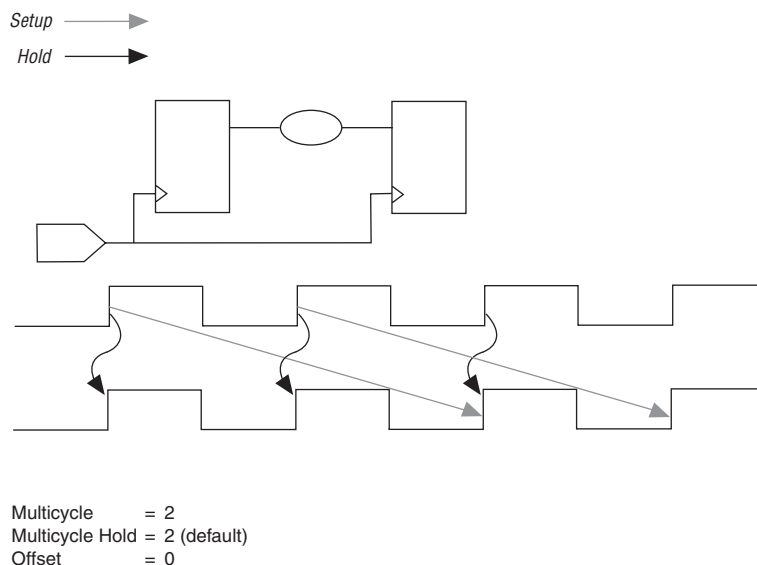


You can assign multicycle paths in your designs to instruct the Quartus II Timing Analyzer to relax its measurements, thus avoiding incorrect setup or hold time violation reports. These assignments are made in the **Assignment Editor** (Assignments menu).

Multicycle Hold Assignment

A **Multicycle Hold** assignment, shown in [Figure 4–16](#), 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.

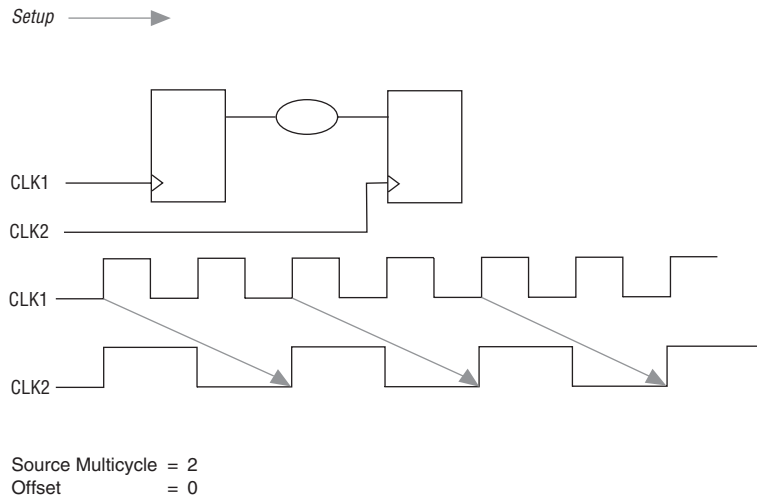
Figure 4-16. Multicycle Hold Assignment



Source Multicycle Assignment

The **Source Multicycle** assignment, shown in [Figure 4-17](#), is useful when the source and destination registers are clocked by related clocks at different frequencies. It is used to extend the required delay by adding periods of the source clock rather than the destination clock.

Figure 4–17. Source Multicycle Assignment



Source Multicycle Hold Assignment

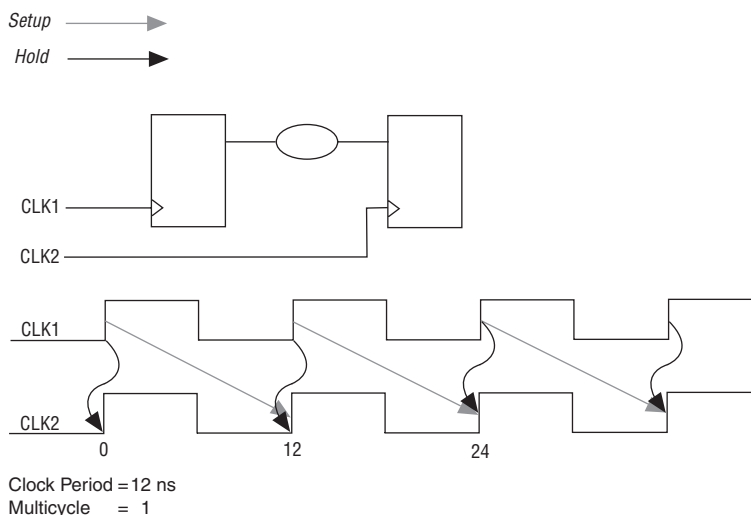
The **Source Multicycle Hold** assignment is useful when the source and destination registers are clocked by related clocks at different frequencies. This assignment allows you to increase the required hold delay by adding source clock cycles.

Typical Applications of Multicycle Assignments

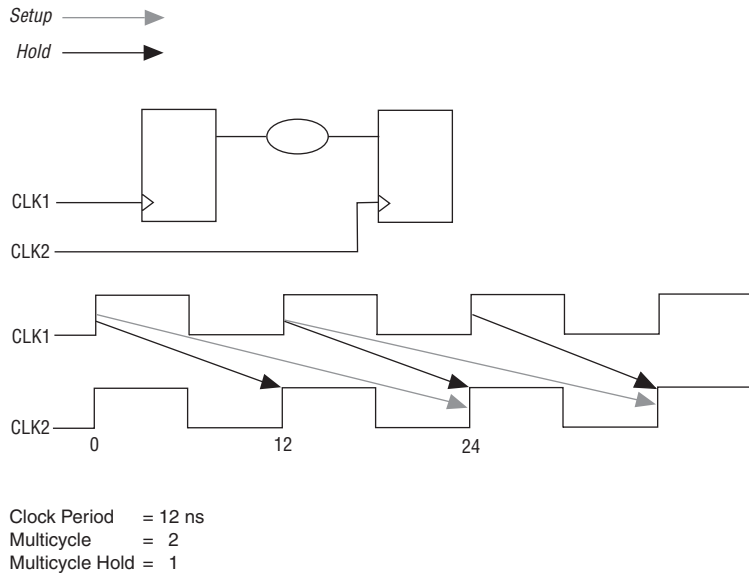
The following examples describe how to use multicycle assignments in your designs.

Simple Multicycle Paths

Figure 4–18 shows the measurement of t_{SU} and t_H for a standard path with a multicycle of 1.

Figure 4–18. t_{SU} and t_H Standard Measurement Paths

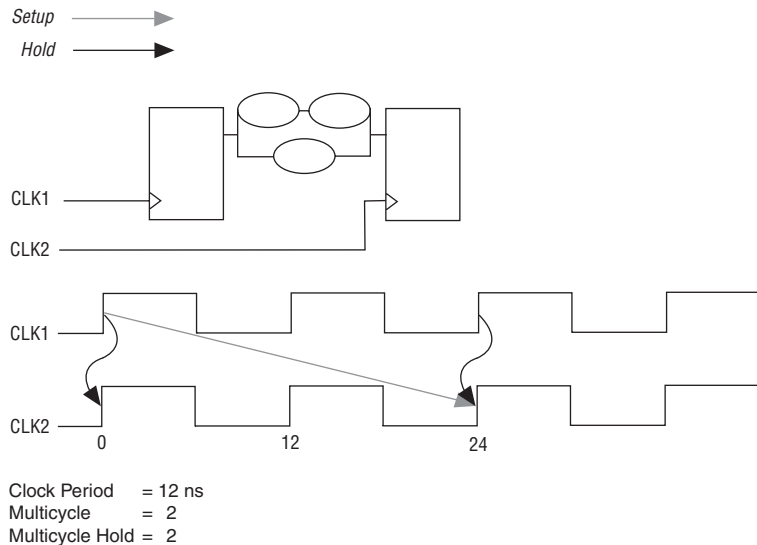
In the example shown in [Figure 4–18](#), both `clk1` and `clk2` have the same period and zero offset. In this figure, where the clocks have a period of 12 ns, the data delay between the source and destination registers must be between 0 ns and 12 ns in order for the circuit to operate. If the data delay 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 4–19. Timing Analysis

In [Figure 4–19](#), 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 4–20](#) 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 4–20. Data Path Delay Example



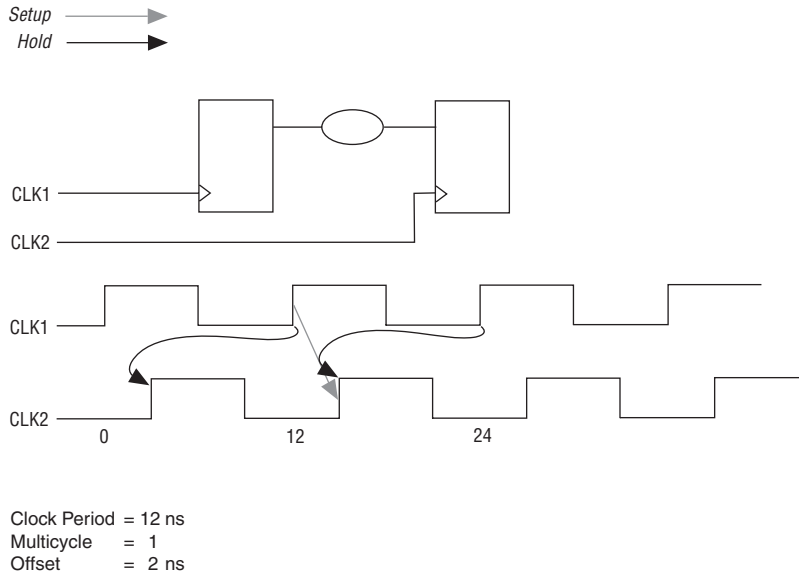
In [Figure 4–20](#), the circuit is intended to operate with a multicycle path of two, however one of the data paths between the registers is less than one clock cycle.

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 circuit operation.

Multicycle Paths with Offsets

In the example shown in [Figure 4-21](#), `clk2` is offset from `clk1` by 2 ns.

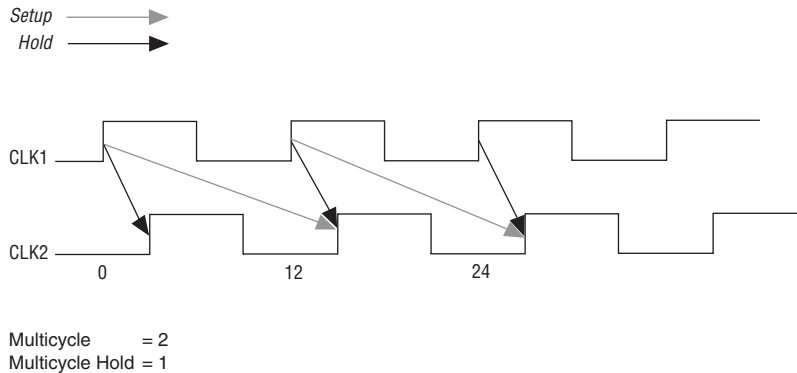
Figure 4-21. Multicycle Paths with Offsets



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, i.e., 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 4-22](#). The circuit operates as a multicycle path of two, assuming the data delay between the registers is between 2 ns and 14 ns.

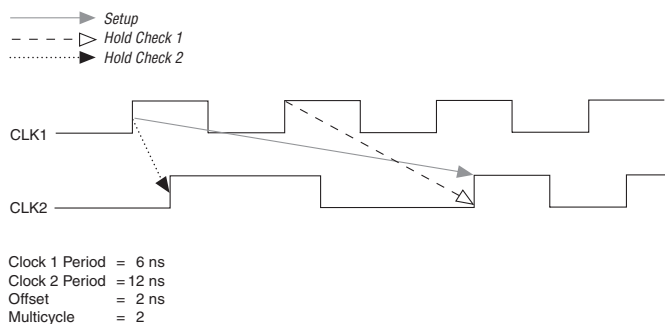
The following Tcl commands can be used to specify the multi-cycle assignments shown in [Figure 4-22](#):

```
set_multicycle_assignment -setup -from clk1 -to clk2 -end 2
set_multicycle_assignment -hold -from clk1 -to clk2 -end 1
```

Figure 4–22. Hold Requirements

Multicycle Paths Across Multi-Frequency Domains

Figure 4–23 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 has to stay 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.

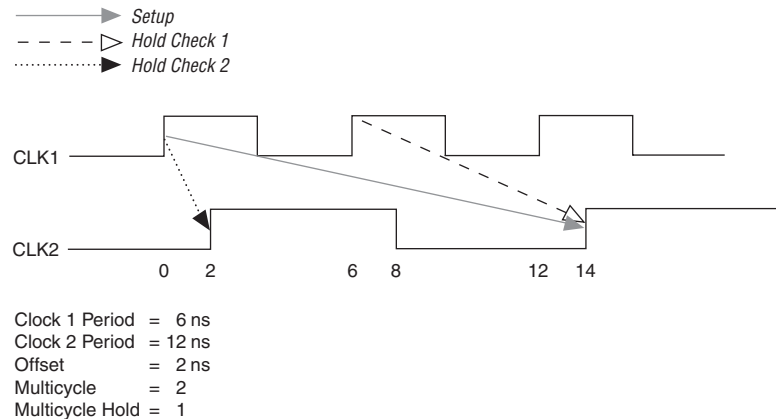
Figure 4–23. Multicycle Hold Checks

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 4-23. 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 4-23.

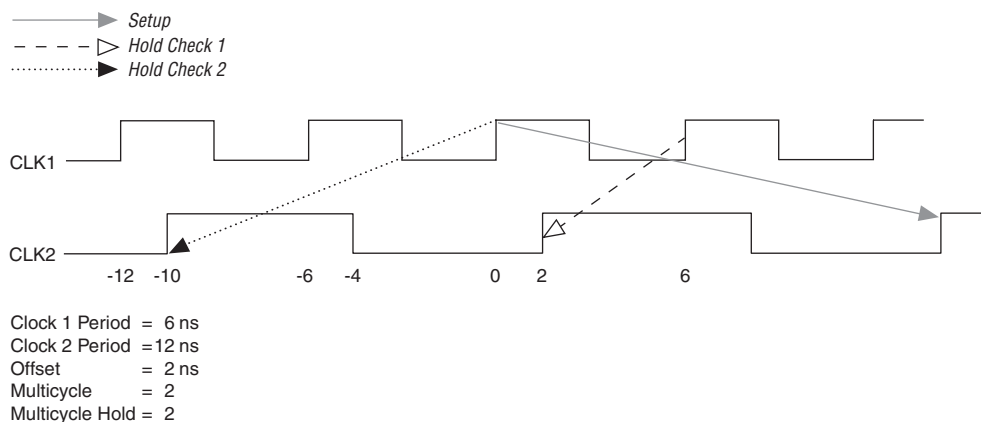
Figure 4-24 illustrates hold time checks for a **Multicycle Hold** assignment of 1.

Figure 4-24. 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 4-25 illustrates hold time checks for the **Default Multicycle Hold** value of 2.

Figure 4–25. 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 ($2 \text{ ns} - 6 \text{ ns}$). The second check, illustrated with the dotted line, requires a minimum data delay of -10 ns ($0 - 10 \text{ ns}$). 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 4–26 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 4–26. Multicycle Hold Checks

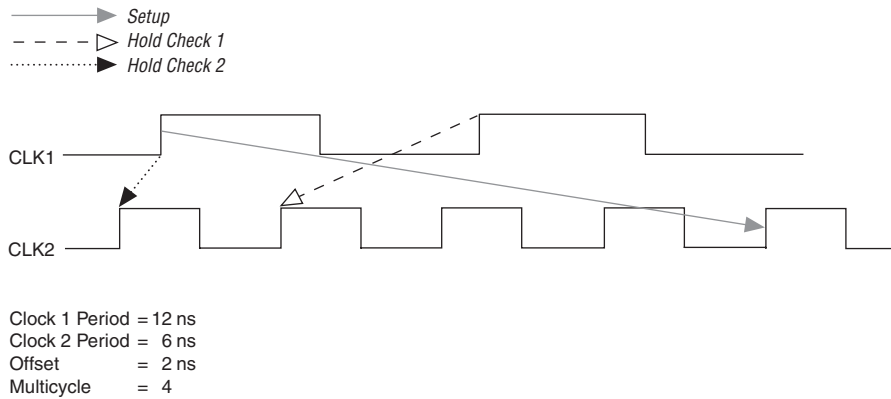
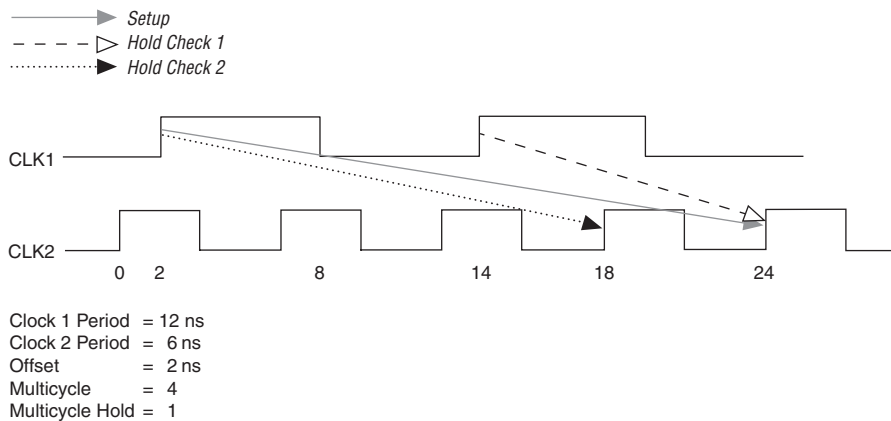


Figure 4–27 illustrates hold time checks for a **Multicycle Hold** assignment of 1.

Figure 4–27. 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.

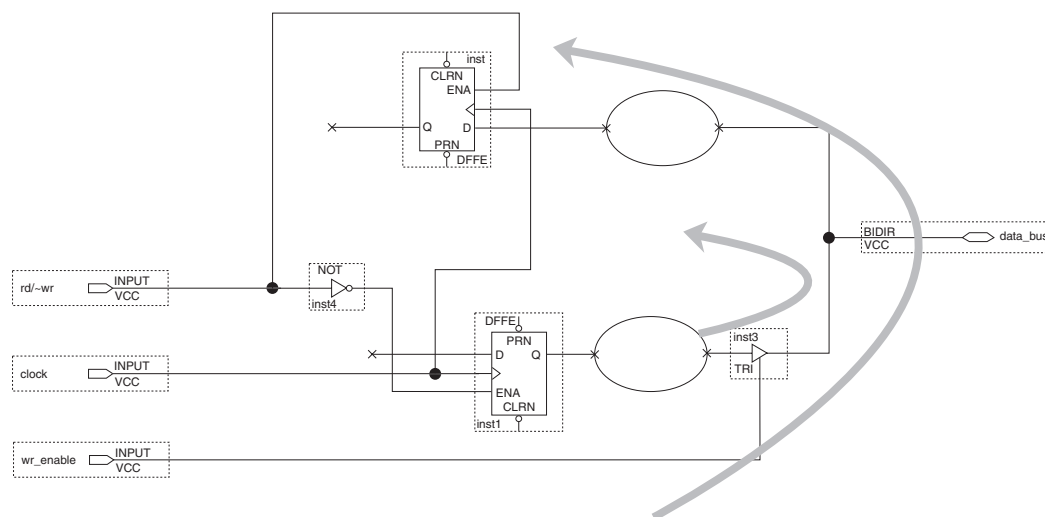
False Paths

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 4-28](#).

Figure 4-28. Cut Off Feedback from I/O Pins

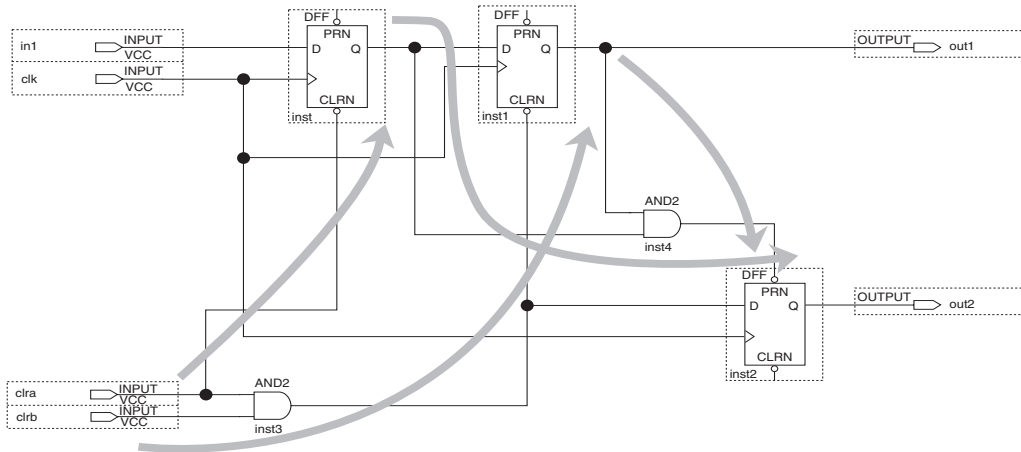


The paths marked with arrows are not measured by timing analysis when this option is turned on. Turn off Cut off feedback from I/O pins to measure these paths during timing analysis.

Cut Off Clear and Preset Signal Paths

This option is turned on by default and cuts the register's clear and preset paths during timing analysis, as shown in [Figure 4-29](#).

Figure 4–29. Cut Off Clear and Preset Signal Paths

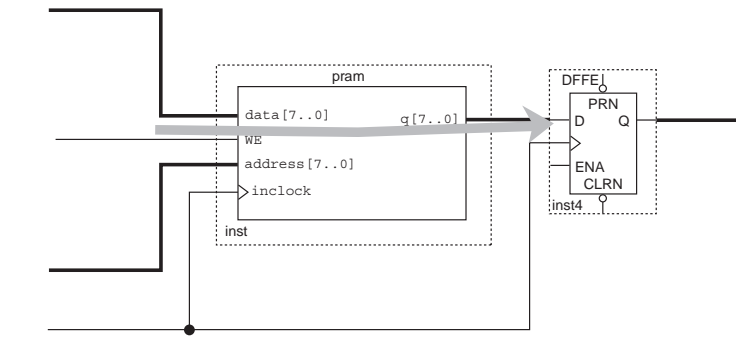


The paths marked with arrows are cut from timing analysis when this setting is turned on. Turn off Cut off clear and preset signal paths to include these paths in the timing analysis report.

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 4–30](#).

Figure 4–30. Cut Off Read During Write Signal Paths

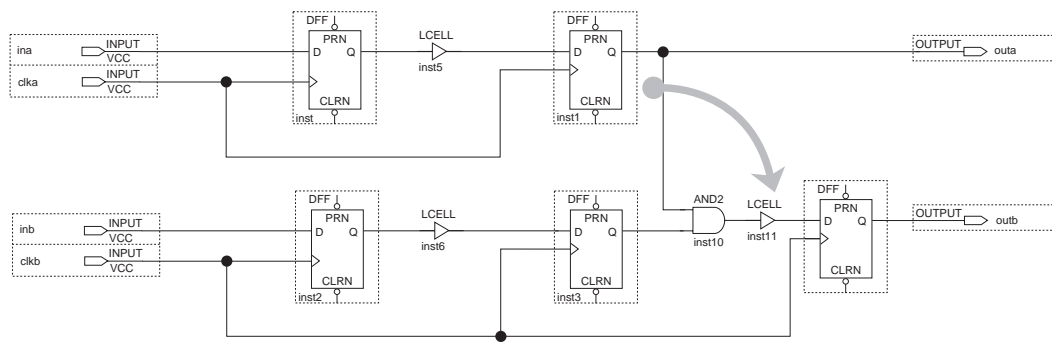


The path marked with an arrow between the `we` input to the memory block `pram` and the register `inst4` is not reported by the Timing Analyzer. This path is reported if Cut off read during write signal paths is turned off.

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 4-31](#).

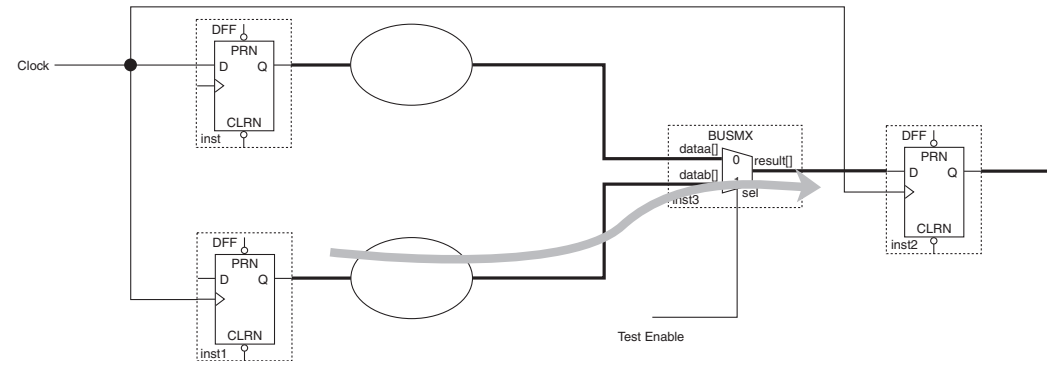
Figure 4–31. Cut Paths Between Unrelated Clock Domains



For the circuit shown in [Figure 4-31](#), 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 4-32](#) shows an example of a false path.

Figure 4–32. False Path Signal

In Figure 4–32, 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, Altera recommends using clock enables instead of gated clocks in your design, although this may not always be 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.

The Quartus II software attempts to meet the following timing requirements on I/O paths by default:

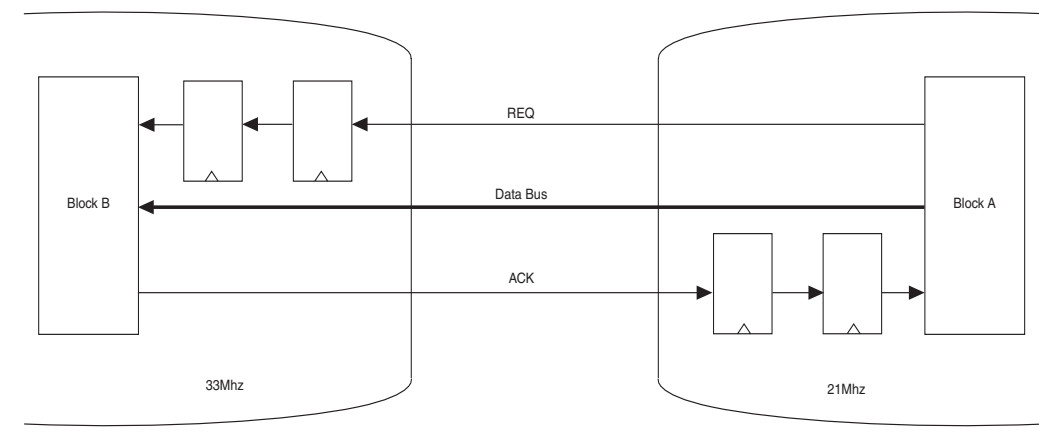
- Hold time (t_H) from I/O pins to registers
- Minimum t_{CO} from registers to I/O pins
- Minimum t_{PD} from I/O pins or registers to I/O pins or registers

You can change the setting to direct the Quartus II software to also attempt to meet register-to-register hold time requirements.

Timing Analysis Across Asynchronous Domains

In cases in which 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 meta-stability. One of the most common techniques used is to enforce a full handshake protocol between the asynchronous boundaries. `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 meta-stability due to asynchronous signals.

Figure 4-33 shows the interaction across asynchronous boundaries.

Figure 4–33. Interaction Across Asynchronous Boundaries

Minimum Timing Analysis

Minimum timing analysis measures and reports minimum t_{CO} , minimum t_{PD} , t_H , and clock hold. Minimum Timing analysis is performed by checking for minimum delay requirements with best-case timing models (delay 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.

Minimum 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 Minimum Timing Analysis

To perform minimum timing analysis with the best-case timing models (delay models), choose **Start > Start Minimum Timing Analysis** (Processing menu). If you use the `quartus_tan` command-line executable, specify the `--min` option. The following tcl example will read the project netlist and generate a Minimum timing report.

```
Quartus_tan --min <project_name>
```

Minimum Timing Analysis Reporting

You can examine the results of minimum timing analysis in the Timing section of the compilation report in the Quartus II GUI. The text-based report generated during timing analysis is called *<project name>.tan.rpt*. The same name is used for the report file generated during regular timing analysis, so that previous timing analysis results is overwritten.

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).

Third-Party Timing Analysis Software

You can also use the PrimeTime software to perform timing analysis. Select **PrimeTime** as the Timing Analysis tool in the Timing Analysis page of the **Settings** dialog box (Assignment menu). The Quartus II Timing Analyzer generates a Verilog or VHDL netlist, a *.sdo* file, and a Tcl script that you can specify in the PrimeTime software to perform timing analysis.

Advanced Timing Analysis & Reports Using Tcl Scripts

Two frequently-used commands are:

- `project_open <project_name>` (To open the project in the project directory)
- `create_timing_netlist` (To generate timing information from a compiled design in the project directory)

`report_timing` command gives you more control over how you want to report your timing analysis results.

```
Usage: 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] [-dqs_read_capture] [-stdout]
[-file <name>] [-append] [-from <names>] [-to <names>]
[-clock_filter <names>] [-longest_paths]
[-shortest_paths] [-all_failures]
```

Examples:

```
report_timing -file <file_name>
```

This command writes out worst timing path, one for each of the t_{su} , t_{hr} , 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 -npaths 2 -file file_name
```

This command writes out 2 timing paths for each of the constraints in **file_name**.

```
report_timing -tsu -npaths 3
```

This command reports 3 worst paths of the t_{su} constraint only.

```
report_timing -clock_filter *_clk0
```

This command will report 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 like ***pll0*_clk0**. These clock names are not limited to absolute or relative clocks defined by the user but also include outputs of the PLLs.

```
report_timing -from in1 -to *utopia*
```

This command will list all timing paths starting from input, **in1**, to any registers or outputs that have **utopia** as part of their name.

```
report_timing -to {out\[4\]}
```

This command will list all timing paths that end at bit 4 of the output bus **out[4:0]**. Back slash has to precede every bracket character and the string has to be enclosed in braces for proper interpretation.

Advanced scripting example1:

```
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
```

This script reports all the registers in a design along with their respective locations on the chip.

Advanced scripting example2:

```

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

```

This script starts with traversing through a list of all the registers in a design by using `get_timing_nodes -type reg` command. The script then uses a `for each` 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 the total clock network delay w.r.t 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.

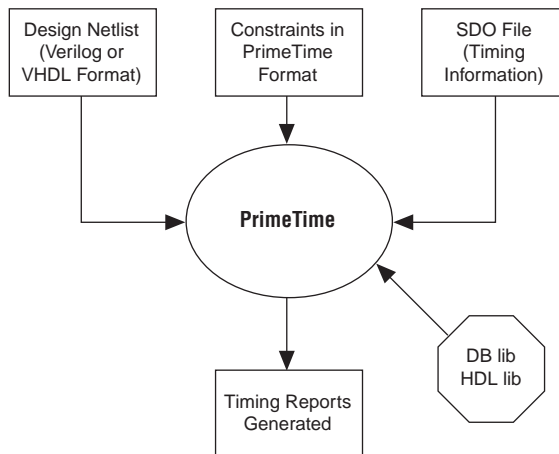
Conclusion

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.

Introduction

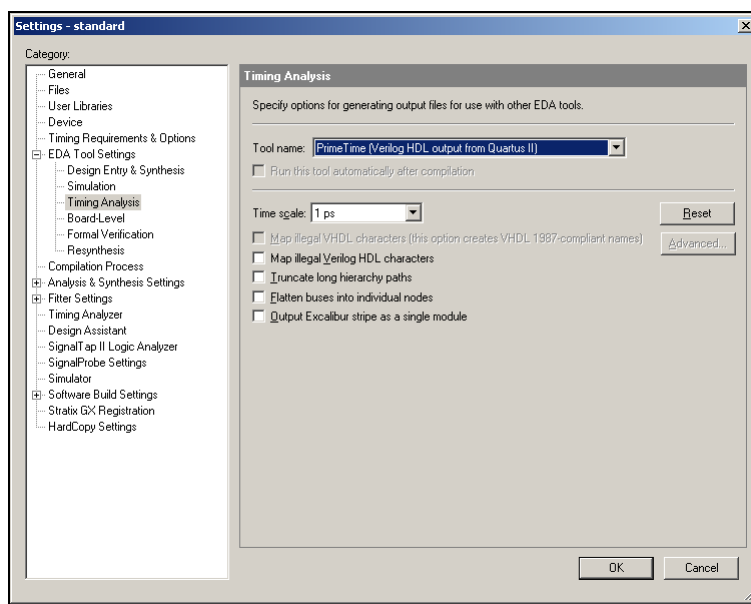
PrimeTime is an industry standard sign-off tool, used to perform static timing analysis on most ASIC designs. The Quartus® II software provides a path to enable users to run PrimeTime on their Quartus designs, exporting netlist, constraints specified in Quartus format, and libraries to the PrimeTime environment. Figure 5–1 shows the PrimeTime flow diagram.

Figure 5–1. PrimeTime Flow Diagram



Quartus II Settings to Generate PrimeTime Files

To set the Quartus II software to generate PrimeTime files, choose **Settings** (Assignments menu). Choose **EDA Tool Settings > Timing Analysis** in the **Category** dialog box to display the **Timing Analysis** window. In the **Timing Analysis** window, click on the **Tool name** pull down menu and select **PrimeTime (Verilog HDL output from Quartus II)** or **PrimeTime (VHDL output from Quartus II)**, as shown in Figure 5–2. This setting enables the Quartus II software to produce three files for the PrimeTime tool, which are then written into the **timing/primetime** directory of the current project.

Figure 5–2. Setting the Quartus II Software to Generate PrimeTime Files

Files Generated for the PrimeTime Environment

This section describes the three files that the Quartus II software creates for the PrimeTime tool.

- `<project_name>.vo` or `<project_name>.vho` files

This is the netlist file written in either Verilog (`.vo`) or VHDL (`.vho`) format, depending on the format selected in the EDA settings. This file contains the flat netlist representing the entire design.

- `<project_name>_v.sdo` or `<project_name>_vhd.sdo` files

These files contain the timing information for each timing arc in the design. Like the netlist files, these files are written in either Verilog (`_v`) or VHDL (`_vhd`) format, depending on the selection made in the EDA settings. This file corresponds to the worst-case delay values of the timing arcs if regular timing analysis is performed in the Quartus II software.

If you want to use the best-case delay values for PrimeTime analysis, you must perform a Minimum Timing Analysis in the Quartus II software. This is a two-step process, as follows.

1. Select **Start > Start Minimum Timing Analysis** (Processing menu).

2. Select **Start > Start EDA Netlist Writer** (Processing menu).

This will create a `<project_name>_v_min.sdo` or `<project_name>_vhd_min.sdo` file, which contains the best-case delay values for each timing arch.



It is up to you to point to either best-case or worst-case delay values during the PrimeTime processing by specifying the appropriate file name in the Tool Command Language (Tcl) script file described below.

- `<project_name>_pt_v.tcl` or `<project_name>_pt_vhd.tcl` files

These files contain the search path to, and the names of, the PrimeTime database library files provided by Altera. A file referred to in this Tcl file (**device_all_pt.v** or **device_all_pt.vhd**) contains the Verilog/VHDL description of each library cell. The search path and link path are defined at the beginning of the Tcl file. The search path must be modified, depending on where these libraries are stored. The link path contains the names of all database files, and it does not need to be modified.

Here is an example of the search path and link path defined in the Tcl file:

```
set quartus_root ". /appsl/altera/quartus/II-3.0"

set search_path [list . $quartus_root
/appsl/altera/quartus/II3.0/eda/synopsys/primetime/lib ]

set link_path [list * stratix_async_io_lib.db
stratix_io_register_lib.db stratix_lvds_receiver_lib.db
stratix_async_lcell_lib.db stratix_lvds_transmitter_lib.db
stratix_core_mem_lib.db stratix_lcell_register_lib.db
stratix_mac_out_internal_lib.db stratix_mac_mult_internal_lib.db
stratix_mac_register_lib.db stratix_memory_register_lib.db
stratix_pll_lib.db alt_vt1.db]

read_verilog stratix_all_pt.v
```

This Tcl file also contains equivalent constraints in PrimeTime format, converted automatically by the Quartus II software from constraints in Quartus II format. Additional PrimeTime commands can be placed in the Tcl file to report on, or analyze, timing paths. This Tcl file also has a command to read the SDO file generated by the Quartus II software. Depending on which SDO file is desired, either with best-case or worst-case delays, the appropriate SDO file name should be specified.

Sample of Constraints Specified in PrimeTime Format

The PrimeTime constraints shown in Table 5–1 are automatically generated by the Quartus II software. The `set_input_delay -max` command is equivalent to the t_{SU} 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 5–1 shows the automatically-generated PrimeTime constraints and their Quartus II software equivalents.

<i>Table 5–1. Equivalent Quartus II & PrimeTime Constraints</i>	
PrimeTime Constraint	Quartus II Equivalent
<code>create_clock -period 10.000 -waveform {0 5.000} [get_ports clk] \-name clk</code>	Clock defined on input pin, clock of 10 ns period 50% duty cycle
<code>set_input_delay -max -add_delay 9.000 -clock [get_clocks clk] \ [get_ports din]</code>	t_{SU} 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_H of 1 ns on input pin, din
<code>set_output_delay -max -add_delay 7.000 -clock [get_clocks clk] \ [get_ports out]</code>	t_{CO} of 3 ns on output pin, out

PrimeTime Timing Reports

This section describes the timing reports that the PrimeTime tool generates, and the Tcl script commands that control each report's contents.

■ `report_timing -nworst 100 > file.timing`

This command, which can be inserted at the end of the Tcl file to report timing paths in PrimeTime, will generate a list of the 100 worst paths, and place this data into a file called **file.timing**.

Timing paths in PrimeTime are listed in the order of most-negative-slack to most-positive-slack. Failing paths are not reported under each constraint's category, as they are in the Quartus II software. 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

This section presents a sample timing report.

Table 5–2. Sample PrimeTime Timing Report		
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.MO (mux21)	0.000	2.524 r
out~I.and2_22.Y (AND2)	0.000	2.524 r
out~I.out_mux1.MO (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

The start point in this report is a register clocked by clock, `clk`. 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 it is in the Quartus II software.

Running PrimeTime

PrimeTime is only available to run on Unix systems. The three files created by the Quartus II software must be transferred to a Unix machine. PrimeTime runs in shell mode by accepting scripts in Tcl format. The `<project_name>_pt_v.tcl` script file, for example, is executed in the following way:

Type the following command at the UNIX command line prompt, and press the Return key:

```
pt_shell -f project_name_pt_v.tcl
```

After all commands in the Tcl script file are executed, "pt_shell>" prompt appears. More `pt_shell` commands can be executed at that prompt, including the following:

- `man report_timing`

This command will list details of how to use the `report_timing` command and all related options.

- `help`

Entering this command at the `pt_shell` prompt lists all the commands available in the `pt_shell`.

- `quit`

Entering this command at the `pt_shell` prompt closes the `pt_shell`.

You can also activate `pt_shell` without a script file by entering `pt_shell` at the UNIX command line prompt.

Conclusion

The Quartus II-generated netlist, constraints, and timing information can be exported into the PrimeTime environment seamlessly. PrimeTime can be used to do worst-case and best-case timing analysis just as in the Quartus II software. PrimeTime timing reports show any violations and slacks.

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, Early Power Estimation](#)
- [Chapter 7, Simulation-Based Power Estimation](#)

Revision History

The table below shows the revision history for [Chapters 6 and 7](#).

Chapter(s)	Date / Version	Changes Made
6	June 2004 v2.0	<ul style="list-style-type: none"> • Updates to tables, figures. • New functionality for Quartus 4.1.
	Feb. 2004 v1.0	Initial release
7	June 2004 v2.0	<ul style="list-style-type: none"> • Updates to tables, figures. • New functionality for Quartus 4.1.
	Feb. 2004 v1.0	Initial release

Introduction

As designs grow larger and processes continue to shrink, power becomes an ever-increasing concern. 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™, Stratix GX, and Cyclone™ device power consumption can be calculated using the Microsoft Excel (Excel)-based power calculator or the Simulation-Based Power Estimation feature in the the Quartus® II software, which is described in the *Simulation-Based Power Estimation* chapter in Volume 3 of the *Quartus II Handbook*.

You can use the Excel-based power calculator during the board design and layout phase to estimate power and design for proper power management. The simulation-based power estimation feature in the Quartus II software (when simulation vectors are available) can verify that your design is within your power budget.

Excel-Based Power Calculator

An Excel-based power calculator, which provides a current (I_{CC}) and power (P) estimation based on typical conditions (room temperature and nominal V_{CC}), is available on the Altera websites for the Stratix, Stratix GX and Cyclone devices, under **Design Utilities**. The power calculator is divided into sections, with each section representing an architectural feature of the device, including 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 power calculator to estimate the device power consumption. The sub-total of the I_{CC} and power consumed by each architectural feature is reported in each section in milliamps (mA) and milliwatts (mW), respectively.

Before reading this chapter, you should be familiar with the Excel-based Stratix, Stratix GX, or Cyclone power calculators available on the Altera website.



For more information about how to use the Excel-based power calculator, see the *Estimating Power in Stratix, Stratix GX, and Cyclone Devices User Guide*.

Figures 6–1 through 6–5 show sections of the Stratix power calculator.

Figure 6–1. Device and I_{CC} Standby Sections in the Stratix Power Calculator

Altera® Stratix™ Device Power Calculator Spreadsheet Version 3.0

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.

Copyright Altera Corporation. All rights reserved.

Comments:

Device

Device	Package	Temperature Grade	Vccint	Total Pwr (mW)	Total Pwr (mW)	Total P (mW)
EPF10K10	100 Pin Quad BGA	C - commercial	1.5 V	450.00	0.00	450.00

Import Data **Enter Toggle %** **Clear All Values**

Icc Standby (mA)

Standby mode: 300

Thermal Analysis

Tj (Degrees C)	Ta (Degrees C)	Required RJA
80	40	80.00

TOTAL

	Icc (mA)	Power (mW)
Internal (Vccint)	300.00	450.00
I/O (Vccio)	0.00	0.00
TOTAL	300.00	450.00

Power-Up Icc (mA)

Maximum: 5000

The power-up current is shown on screen and is not added to the total current because it is independent of the current consumption during device start mode. For more information about power-up current in the device, see the power consumption section in: http://www.altera.com/literature/stratix/stratix_4_vol_1.pdf

Figure 6–2. Clock Network Section in the Stratix Power Calculator

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 6–3. Logic Elements Section in the Stratix Power Calculator

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 6–4. RAM Blocks Section in the Stratix Power Calculator

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 _{ccint} (mA)	P _{int} (mW)
1	100	9	9	12.50	50	Single-Port	8.47	2.73	11.20	16.90
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 _{ccint} (mA)	P _{int} (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 _{ccint} (mA)	P _{int} (mW)
1	100	48	48	12.50	2	Two-Port	2.55	0.19	2.74	4.11

Figure 6–5. General I/O Power Section in the Stratix Power Calculator

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/VCMOS_24	SDR	42.55
4	66	128	12.50	10	3.3-LVTTL	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, you can either use the Excel-based power calculator or the simulation-based power estimation feature in Quartus II.

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 can happen before the FPGA design is complete. If the FPGA design has not yet

begun, or is not complete, an estimate of the power consumption for the design can be made using the Excel-based power calculator. [Table 6–1](#) shows the power estimation flow when using the Excel-based power calculator when the FPGA design has not begun.

Table 6–1. Power Estimation Before FPGA Design Has Begun

Steps to Follow	Advantages	Disadvantages
1. Download the Excel-based power calculator from the Altera website	Power Estimation can be done before any FPGA design is complete	Accuracy is dependent on user input and estimate of the device resources
2. Manually fill in the power calculator		Can be time consuming

When the FPGA design is partially complete, the power estimation file generated by the Quartus II software can help to fill in the Excel-based power calculator. After using the Import Data macro to import the power estimation file information into the Excel-based power calculator, you can edit the power calculator to reflect the device resource estimates for the final design.



For more information about how to generate the power estimation file in the Quartus II software, see [“Quartus II Power Report File” on page 6–6](#). For more information about how use the Import Data macro to import the power estimation file information into the Excel-based power calculator, see the *Estimating Power in Stratix, Stratix GX, and Cyclone Devices User Guide*.

Table 6–2 shows the power estimation flow for the Excel-based power calculator when the FPGA design is partially complete.

Table 6–2. 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 Provides the flexibility to automatically fill the power-calculator based on results of compilation in the Quartus II software	Accuracy is 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 Excel-based power calculator from the Altera website		
4. Run the import data macro to automatically populate the Excel-based power calculator		
5. Optionally, edits to the power calculator can be made to reflect the device resources used in the final design		

When the FPGA design is complete, the device power consumption can be estimated with the simulation-based power estimation feature in Quartus II. The Quartus II Simulator provides simulation-based power estimation for Stratix, Stratix GX, Cyclone, HardCopy™ Stratix, MAX® 7000AE, MAX 7000B, and MAX 3000A devices. To use the power estimation feature, you must provide a Vector Waveform File (.vwf) or Power Input File (.pwf) to the Quartus II Simulator and perform a timing simulation.



For more information about how to use the simulation-based power estimation feature in the Quartus II software, see the *Simulation-Based Power Estimation* chapter in Volume 3 of the *Quartus II Handbook*.

Table 6–3 shows the power estimation flow for the simulation-based power estimation feature in the Quartus II software when the FPGA design is complete.

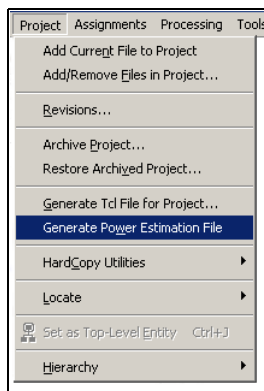
<i>Table 6–3. Power Estimation When FPGA Design Is Complete</i>		
Steps to follow	Advantages	Disadvantages
1. Compile the FPGA design in Quartus II	Provides the most accurate power estimation since the simulation stimuli reflect actual device behavior	Power Estimation done later in the FPGA design cycle
2. Create the stimulus for simulation		
3. Simulate the design using Quartus II vector files or a Power Input File (.pwf) from a third party simulation tool		
4. Quartus II Simulator reports the power estimation results		

Quartus II Power Report File

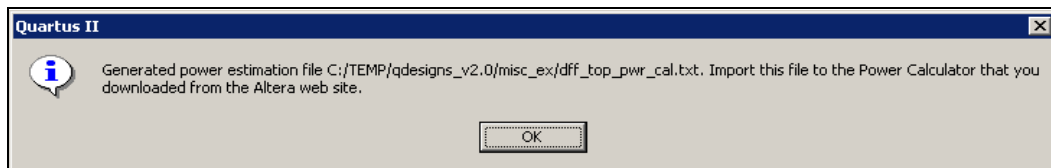
When filling out the Excel-based power calculator, you enter the device resources, operating frequency, toggle rates and other parameters in the power calculator. This requires familiarity with the design. If you do not have an existing design, then you must estimate the number of device resources used in your design.

If you already have an existing design or a partially completed design, the power estimation report file that is generated by the Quartus II software version 4.1 can aid in filling out the power calculator.

To generate the power estimation file, you must first compile your design in the Quartus II software version 4.1. After compilation is complete, choose **Generate Power Estimation File** (Project menu), which instructs the Quartus II software to write out a power estimation report text file. See Figure 6–6.

Figure 6–6. Generate Power Estimation File Option

After the Quartus II software successfully generates the power estimation report file, a message will be displayed. See [Figure 6–7](#).

Figure 6–7. Generate Power Estimation File Message

The power estimation report file is named *<name of Quartus II project>_pwr_cal.txt*. [Figure 6–8](#) is an example of the contents of a power estimation file generated by the Quartus II software version 4.1.

Figure 6–8. 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=iodata_rate_GIO1 value=SDR>
```

The Stratix Power Calculator v3.0, Stratix GX Power Calculator v1.3, and Cyclone Power Calculator v1.2 power calculation spreadsheets include the Import Data macro that parses the information in the power estimation file and transfers it into the Excel-based power calculator. If you do not want to use the macro, you can also transfer the data into the Excel-based power calculator manually.

If your existing Quartus II project represents only a portion of your full design, you should manually enter in the additional resources that are used in the final design. Therefore, after importing the power estimation file information into the Excel-based power calculator, you can edit it to add in additional device resources.



For completed designs, see the *Simulation-Based Power Estimation* chapter in Volume 3 of the *Quartus II Handbook*.

Conclusion

The power calculator is an easy and useful tool to estimate the power consumption for your designs based on typical conditions. The power estimation file generated by the Quartus II software helps to fill in the Excel-based power calculator available on the Altera website. Board-level and FPGA designers can benefit from the power estimation report file generated by the Quartus II software to more accurately estimate power.

References

Estimating Power in Stratix, Stratix GX, and Cyclone Devices User Guide

Introduction

After completing the design, synthesis, and place-and-route steps in the design cycle, you should use the Simulator in the Quartus® II software to perform a simulation to verify design functionality. The simulation should include a Simulation-based power estimation. The power estimation provides an accurate way to estimate the power consumed by your design because it is based on the simulation stimuli that reflects the actual design behavior. In addition to providing design verification, the Simulator supports simulation-based power estimation for Stratix™, Stratix GX, Cyclone™, HardCopy Stratix™, MAX⁶ 7000AE, MAX 7000B, and MAX 3000A devices.

Since simulation typically happens later in the design cycle, simulation-based power estimation is generally used to verify the power consumption of a device already on board. However, simulation-based power estimation is also a useful tool to estimate power in portions of a larger design when integrating smaller designs into larger FPGAs.

The device power consumption can be estimated before the simulation stage. To use the power estimation feature, you must provide a Vector Waveform File (.vwf) or Power Input File (.pwf) to the Quartus II Simulator and perform a timing simulation.



For more information about how to perform an early power estimation of your design, see the *Early Power Estimation* chapter in Volume 3 of the *Quartus II Handbook*.

This chapter explains how to use the simulation-based power estimation feature in the Quartus II software to estimate device power consumption.



It is important to remember that these results should only be used as an estimation of power, not as a specification. The total device current should be verified during device operation as this measurement is sensitive to the actual implementation in the device and to the environmental operating conditions.

Power Estimation in the Quartus II Software



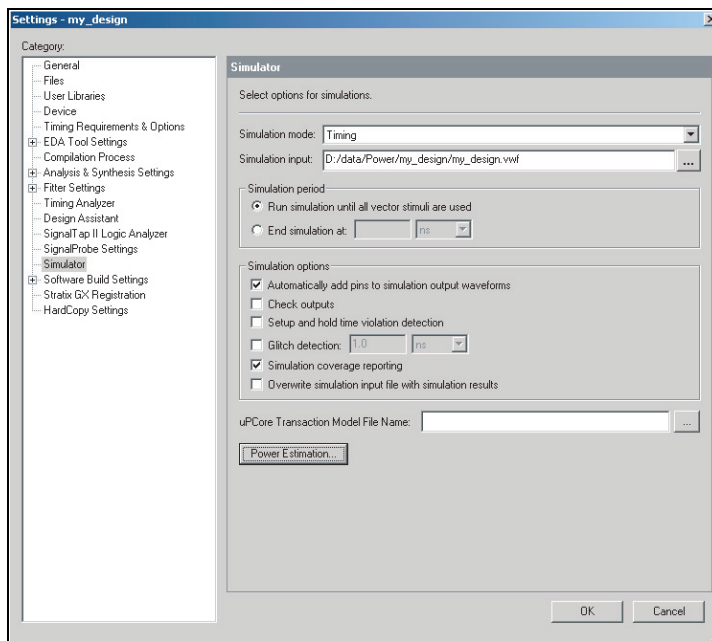
The Quartus II Simulator has a power estimation feature that uses your design simulation vector files to estimate the device power consumption based on typical device-operating conditions. This feature enables you to identify and optimize system-level power consumption in the design cycle.

For more information about how to perform simulations in the Quartus II software, see Quartus II Help.

The power estimation is based on simulation vectors entered in the VWF or VEC and is estimated when performing a timing simulation. To turn on the power estimation feature, follow the steps below:

1. Choose **Settings** (Assignment menu).
2. In the **Settings** dialog box, under the **Category** list, select **Simulator** (see Figure 7–1).

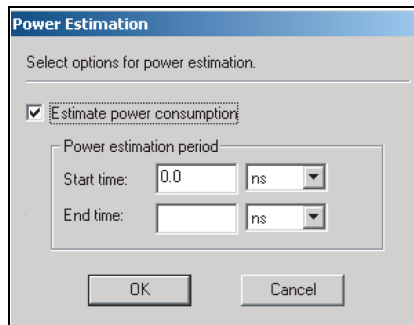
Figure 7–1. Simulator Settings



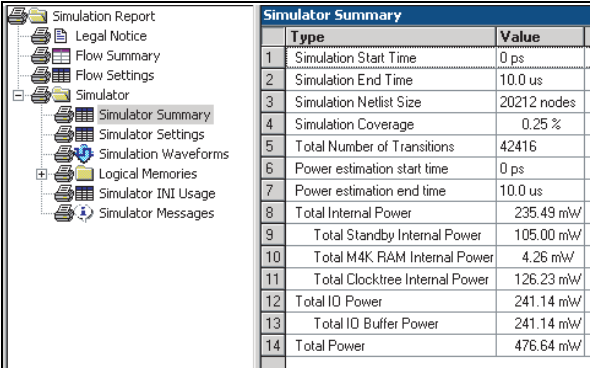
3. In the **Simulator Settings** window, select **Timing** in the **Simulation mode** list.

4. Click **Power Estimation** to open the **Power Estimation** window.
5. In the **Power Estimation** dialog box, turn on the **Estimate power consumption** (see [Figure 7–2](#)). The Simulator calculates and reports the internal power, I/O power, and total power (in mW) consumed by the design during the simulation period.
6. Power estimation can be performed for the entire simulation time, or for a portion of the entire 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. You can specify the start time and end time in the **Power Estimation** dialog box under **Power estimation period**. If no power estimation end time is specified, power estimation ends at the simulation end time.

Figure 7–2. Power Estimation Window



7. After the timing simulation is performed, the estimated power consumption for your design is reported in the Summary section of the Simulation Report. The Simulator Reports the Total Power which is the sum total of Total Internal Power and the Total I/O power. The internal power includes the internal standby power and dynamic power. In the example shown in [Figure 7–3](#) the M4K RAM and the clocktree components contribute to the dynamic power consumed by the design.

Figure 7–3. Simulator Summary


Simulator Summary		
	Type	Value
1	Simulation Start Time	0 ps
2	Simulation End Time	10.0 us
3	Simulation Netlist Size	20212 nodes
4	Simulation Coverage	0.25 %
5	Total Number of Transitions	42416
6	Power estimation start time	0 ps
7	Power estimation end time	10.0 us
8	Total Internal Power	235.49 mW
9	Total Standby Internal Power	105.00 mW
10	Total M4K RAM Internal Power	4.26 mW
11	Total Clocktree Internal Power	126.23 mW
12	Total IO Power	241.14 mW
13	Total IO Buffer Power	241.14 mW
14	Total Power	476.64 mW

Simulation-based power estimation reports a more accurate toggle percentage of your design since it calculates the toggle rate based on the simulation waveforms you provide. Hence, the power estimated by the Quartus II Simulator is more accurate than the Microsoft **excel**-based power calculator. The power calculator is explained in the *Early Power Estimation* chapter in Volume 3 of the *Quartus II Handbook*. It is important to remember that Simulator power results can be only as accurate as the simulation waveforms you provide. To achieve the most accurate results, your simulation waveforms should mimic the behavior of your design.

Estimating Power with EDA Simulation Tools

You can use other EDA simulation tools, such as Model Technology™ ModelSim® software to perform a simulation that includes power estimation data. To do this, you must instruct the Quartus II software to include power estimation data in the Verilog Output File (.vo) or VHDL Output File (.vho). When you are performing a simulation in another EDA simulation tool, the tool uses the power estimation data to generate a Power Input File (.pwf). The PWF file is used in the Quartus II software to estimate the power consumption of your design.



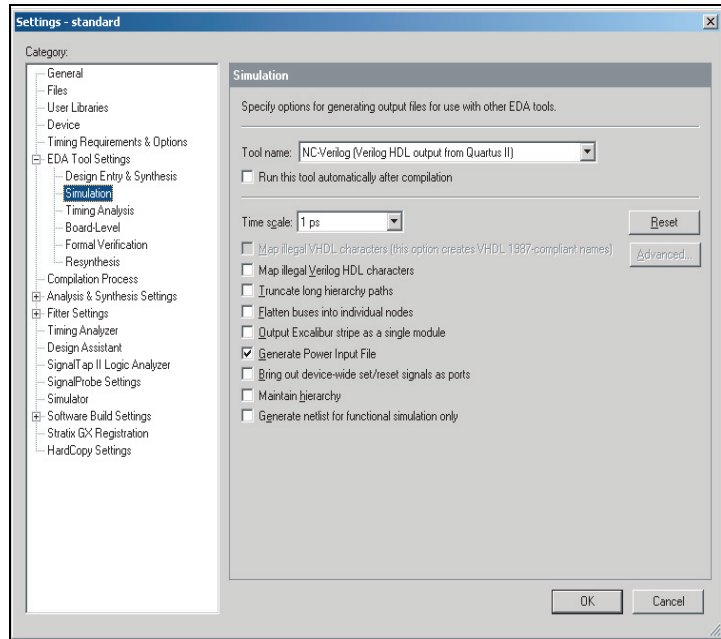
For more information about how to perform simulations in other EDA simulation tools, see the relevant documentation for that tool.

To perform power estimation using the Quartus II software and other EDA simulation tools, follow the steps below:

1. Choose **EDA tool settings** (Assignments menu).
2. In the EDA tools **Settings** dialog box, under the **Category** list, open **EDA Tool Settings** and select **Simulation**.

3. In the **Simulation** dialog box, choose the appropriate EDA simulation tool from the **Tool name** list.
4. Turn on **Generate Power Input File** (see [Figure 7-4](#)).

Figure 7-4. EDA Tool Settings Window



5. Compile the design in the Quartus II software.
6. Perform a timing simulation with the other EDA simulation tool.

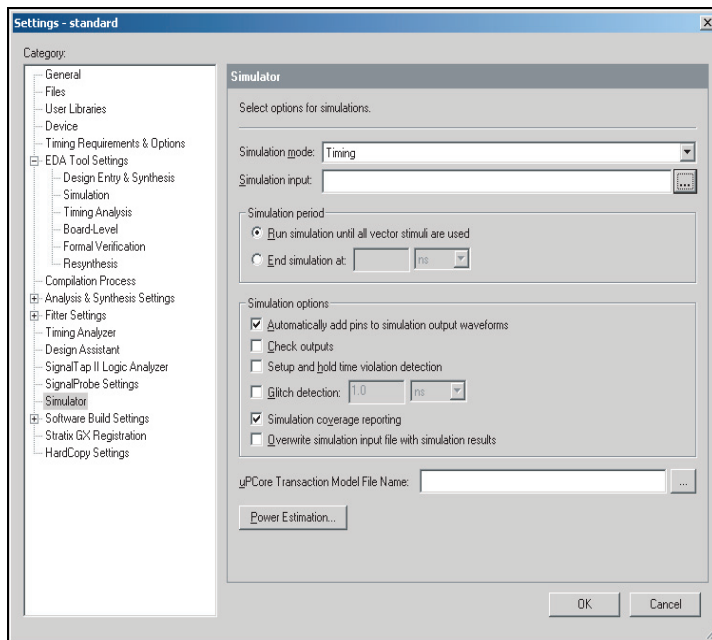
The simulation tool generates the PWF file and places it in the project directory.

7. In the Quartus II software, choose **Settings** (Assignment menu).
8. In the **Settings** dialog box, under the **Category** list, open **Fitter Settings** and select **Simulator**.
9. In the **Simulator** window, select **Timing** in the **Simulation mode** list.

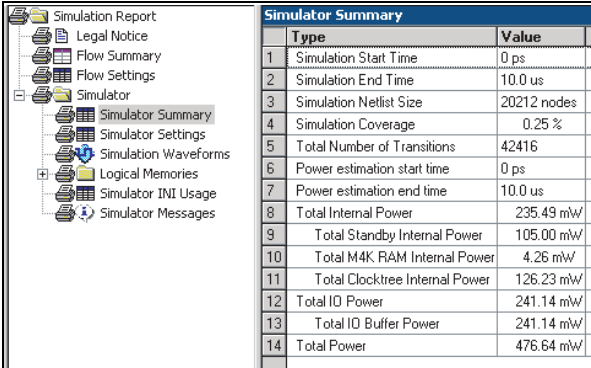
10. Specify the PWF file in the **Simulation input** box (see [Figure 7-5](#)).

You can browse to the appropriate PWF file by clicking the **Browse (...)** button.

Figure 7-5. Simulator Settings Dialog Box



11. In the Quartus II software, perform a timing simulation of your design.
12. View the estimated power consumption in the Simulator Summary section of the Simulation Report (see [Figure 7-6](#)).

Figure 7–6. Simulator Summary


Simulator Summary		
	Type	Value
1	Simulation Start Time	0 ps
2	Simulation End Time	10.0 us
3	Simulation Netlist Size	20212 nodes
4	Simulation Coverage	0.25 %
5	Total Number of Transitions	42416
6	Power estimation start time	0 ps
7	Power estimation end time	10.0 us
8	Total Internal Power	235.49 mW
9	Total Standby Internal Power	105.00 mW
10	Total M4K RAM Internal Power	4.26 mW
11	Total Clocktree Internal Power	126.23 mW
12	Total IO Power	241.14 mW
13	Total IO Buffer Power	241.14 mW
14	Total Power	476.64 mW

Scripting Support

You can run the procedures and make the 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 and examples on Quartus II scripting support, refer to the *Tcl Scripting* and *Command-Line Scripting* chapters in Volume 2 of the *Quartus II Handbook*.

Simulation-Based Power Estimation Settings

Use the following Tcl command to turn on the power estimation feature:

```
set_global_assignment -name ESTIMATE_POWER_CONSUMPTION ON
```

For more information on power estimation settings, refer to [“Power Estimation in the Quartus II Software”](#) on page 7–2.

Use the following Tcl commands to set the power estimation start and end times. Specify the start and end times with quotes, such as “100 ns” for `start_time` and `end_time`:

```
set_global_assignment -name POWER_ESTIMATION_START_TIME "<start_time>"
```

```
set_global_assignment -name POWER_ESTIMATION_END_TIME "<end_time>"
```

Generate a Power Input File

Use the following Tcl command to cause the Quartus II software to generate a PWF for use with third-party EDA simulation software. For more information on estimating power with EDA simulation tools, refer to [“Estimating Power with EDA Simulation Tools”](#) on page 7–4.

```
set_global_assignment -name EDA_GENERATE_POWER_INPUT_FILE ON -section_id eda_simulation
```

Use the following Tcl command to specify the PWF to be used as an input by the Quartus II software to estimate the power consumption of the design.

```
set_global_assignment -name VECTOR_INPUT_SOURCE <file name>.pwf
```

Conclusion

The simulation-based power estimation feature in the Quartus II software is an easy and useful tool to estimate the power consumption for your designs, based on typical conditions. You can use this feature in the Quartus II software and other EDA simulation tools to estimate power and verify that their design is within their power budget.

References

- *Estimating Power in Stratix, Stratix GX, and Cyclone Devices 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, Quick Design Debugging Using SignalProbe](#)
- [Chapter 9, Design Debugging Using the SignalTap II Embedded Logic Analyzer](#)
- [Chapter 10, Design Analysis and Engineering Change Management with Chip Editor](#)
- [Chapter 11, In-System Updating of Memory & Constants](#)

Revision History

The table below shows the revision history for [Chapters 8 to 11](#).

Chapter(s)	Date / Version	Changes Made
8	June 2004 v2.0	<ul style="list-style-type: none">• Updates to tables, figures.• New functionality for Quartus 4.1.
	Feb. 2004 v1.0	Initial release.
9	June 2004 v2.0	<ul style="list-style-type: none">• Updates to tables, figures.• New functionality for Quartus 4.1.
	Feb. 2004 v1.0	Initial release.
10	June 2004 v2.0	<ul style="list-style-type: none">• Updates to tables, figures.• New functionality for Quartus 4.1.
	Feb. 2004 v1.0	Initial release.
11	Aug. 2004 v1.1	Minor typographical corrections.
	June 2004 v1.0	Initial release.

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®, Stratix GX, Cyclone™, APEX™ II, APEX 20KE, APEX 20KC, APEX 20K, and Excalibur™ devices.



You can accomplish the same functionality with the Chip Editor as 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 reserved 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 prior to initial compilation.
2. After initial compilation, determine which nodes you want to route to the reserved SignalProbe pins.
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. Understand the results of the SignalProbe compilation.

Reserving SignalProbe pins

You can reserve an unused pin as a SignalProbe pin before you route an internal signal out of your device. You can reserve your SignalProbe pins before or after a compilation. To ensure that a pin is available for your SignalProbe pin and not to another unassigned user I/O pin, reserve the SignalProbe pin before a compilation.

You may only need a few SignalProbe pins, since you can easily reassign different sources to your SignalProbe pins.

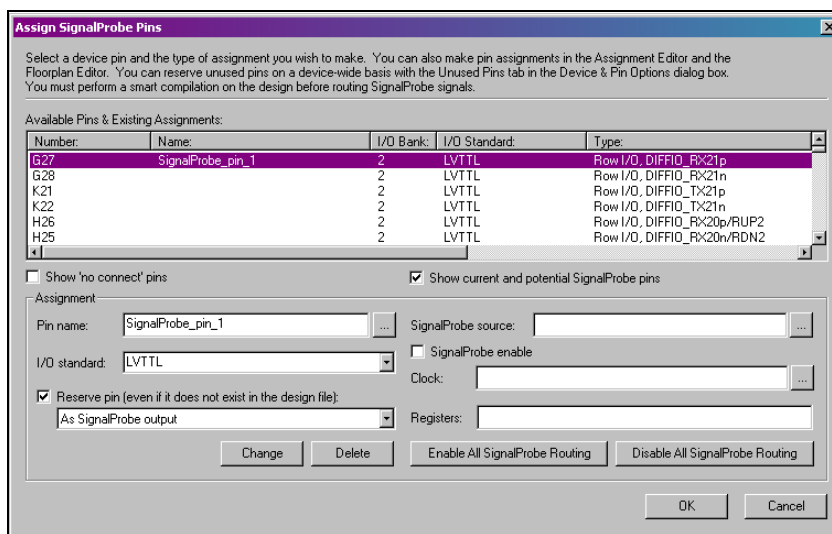
To reserve an unused I/O pin as a SignalProbe pin, perform the following steps:

1. Click **Assign SignalProbe Pins** on the **SignalProbe Settings** page of the **Settings** dialog box (Assignment menu). See [Figure 8–1](#).
2. Turn on **Show current and potential SignalProbe pins** in the **Assign SignalProbe Pins** dialog box.
3. Select a pin **Number** from the **Available Pins & Existing Assignments** list.
4. Type your SignalProbe pin name into the **Pin name** box.
5. Select **As SignalProbe output** from the **Reserve pin** list.
6. Turn on **Reserve pin**.
7. Click **Add** for a new SignalProbe pin.

or

Click **Change** for an existing SignalProbe pin.

8. Click **OK**.

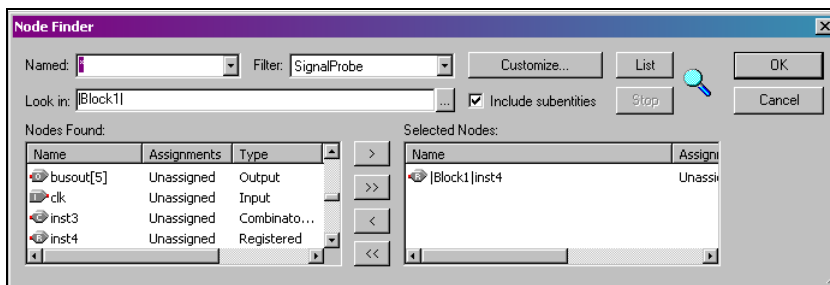
Figure 8–1. Reserving a Pin for SignalProbe in the Assign SignalProbe Pins Dialog Box

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, an unused output pin, or a reserved 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 **Available Pins & Existing Assignments** list, select the pin number for the pin to which you want to add a SignalProbe source. The pin must be a reserved SignalProbe pin, an unused output pin, or a reserved output pin.
3. Browse to a **SignalProbe source**.

The **Node Finder** dialog box appears when you click **Browse** and automatically selects **SignalProbe** in the **Filter** list (see [Figure 8–2](#)). 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 8–2. Available SignalProbe Sources in the Node Finder

- Click **Add** for a new SignalProbe pin.

or

Click **Change** for an existing SignalProbe pin.

- 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 in the **Assignment** box in the **Assign Pins** dialog box.
- 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 respect to a clock and control the latency. The SignalProbe incremental routing feature automatically inserts the number of registers specified in 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}). 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 clock to output delays (t_{CO}).

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 **Available Pins & Existing Assignments** list, select the pin number for the SignalProbe output pin you want to register.
3. Under **Assignment**, type a new **Clock name** in the **Clock box**.
4. Under **Assignment**, type the number of registers necessary to pipeline your SignalProbe source in the **Register box**.



Altera® strongly recommends using global clock signals to clock the added registers.

The MAX II, Stratix, Stratix GX, and Cyclone devices support adding registers to a SignalProbe pin.

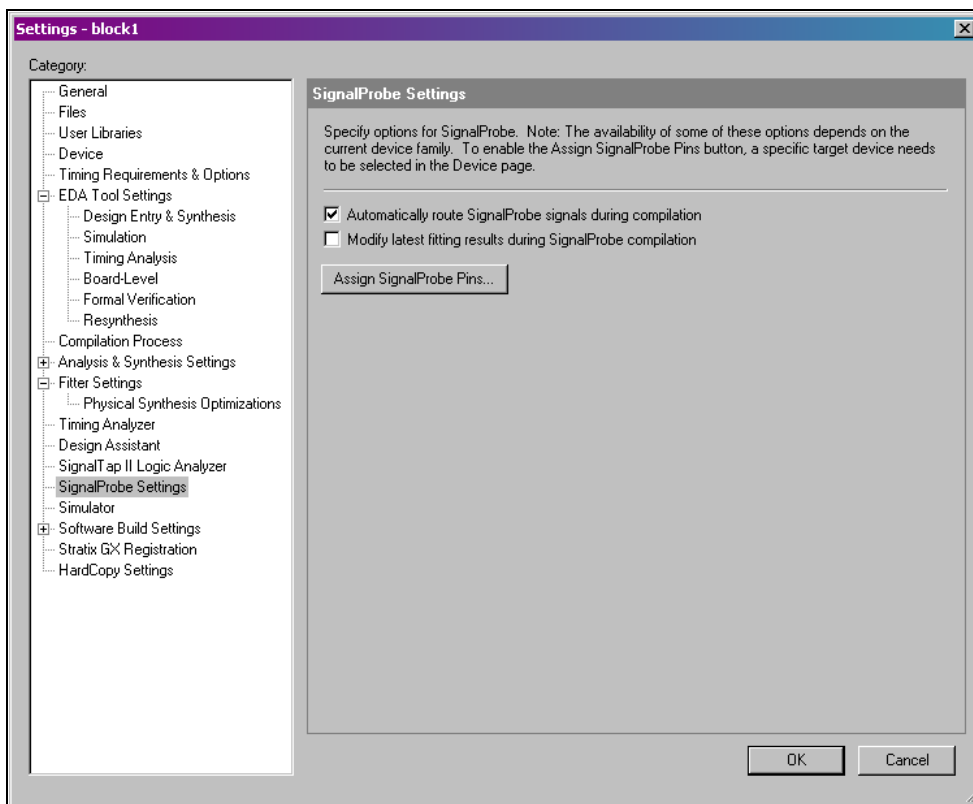
Performing a SignalProbe Compilation

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 8-3](#)).

Figure 8-3. SignalProbe Settings Page in the Settings Dialog Box



To run a SignalProbe compilation manually after a full compilation, choose **Start SignalProbe Compilation** (Processing 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 **Assignment** box in the **Assign SignalProbe Pins** dialog box. You can also enable or disable all

SignalProbe pins by clicking **Enable All SignalProbe Routing** and **Disable All SignalProbe Routing** respectively in the **Assignment** box in the **Assign SignalProbe Pins** dialog box.

The **Enable All SignalProbe Routing** and **Disable All SignalProbe Routing** options are disabled until you turn on **Show current and potential SignalProbe pins** in the **Assign SignalProbe Pins** dialog box.

Running SignalProbe with Smart Compilation

Smart compilation reduces compilation times 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 will get the following message:

```
Error: Can't perform SignalProbe compilation because
design requires a full compilation.
```



Altera recommends turning on smart compilation so that you are always working with the latest settings and design files.

Understanding SignalProbe Routing Failures

If the SignalProbe compilation starts and fails, it could be because of one of the following reasons:

- The SignalProbe compilation failed to find a route from the SignalProbe source to the SignalProbe pin because of routing congestion
- You entered a SignalProbe source that does not exist or is an invalid SignalProbe source.
- The output pin selected is found to be unusable.

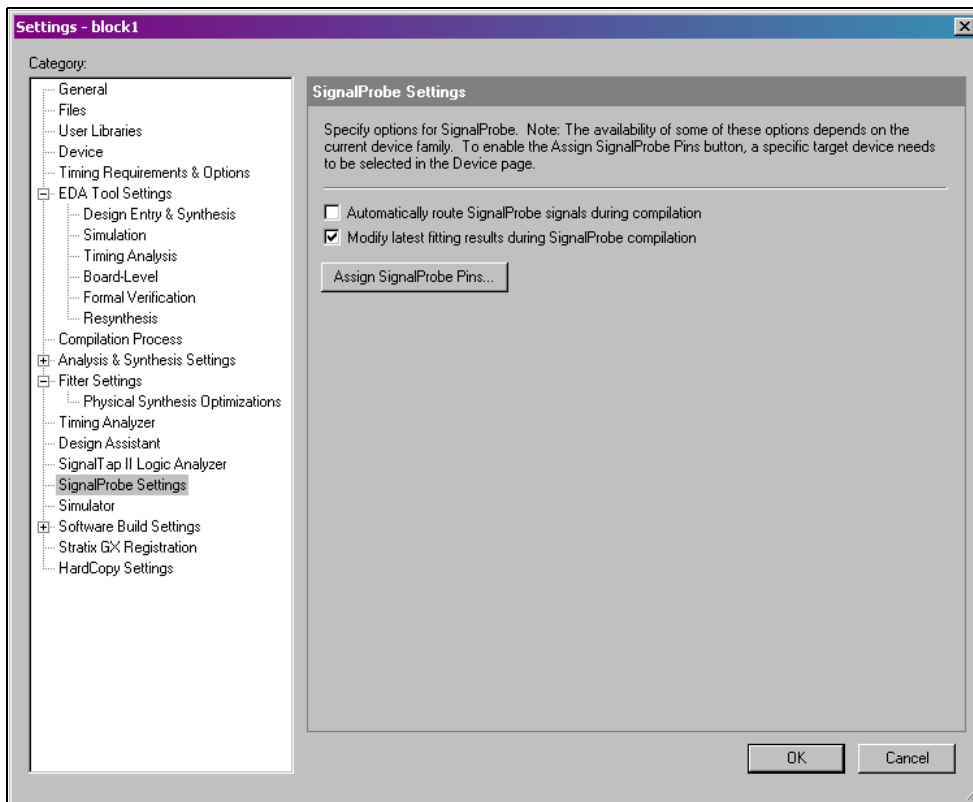
Routing failures can occur if the SignalProbe pin's I/O standard conflicts with other I/O standards in the same I/O Bank.

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 8-4](#)). 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 8–4. SignalProbe Settings Page in the Settings Dialog Box



Understanding the Results of a SignalProbe Compilation

Use the Messages window to view the results of the SignalProbe compilation. This window lists successfully routed SignalProbe pins. In addition, it displays slack information for each successfully routed SignalProbe pin.

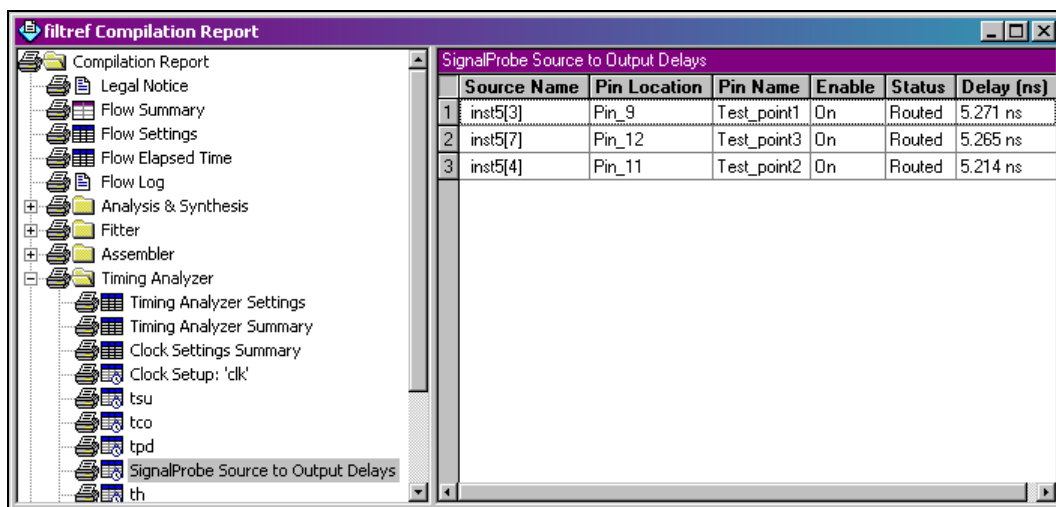
You can view the status and delays of each SignalProbe pin by viewing the **Status** column in the **Assign SignalProbe Pins** dialog box. Table 8–1 describes the possible values for the **Status** column.

Table 8–1. 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

You can find source to output delays for each routed SignalProbe pin in the **SignalProbe Source to Output Delays** page under **Timing Analyzer** in the **Compilation Report** window (see Figure 8–5).

Figure 8–5. SignalProbe Source to Output Delays Page in the Compilation Report Window



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 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 8–2](#).

```
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 8–3](#). 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 8–4](#).

```
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 8-4](#).

```
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 8-5](#).

```
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 8-5](#).

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 8-5](#). 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 8–7](#).

```
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 [“Understanding SignalProbe Routing Failures” on page 8–7](#).

```
set_global_assignment -name SIGNALPROBE_ALLOW_OVERUSE ON
```

Conclusion

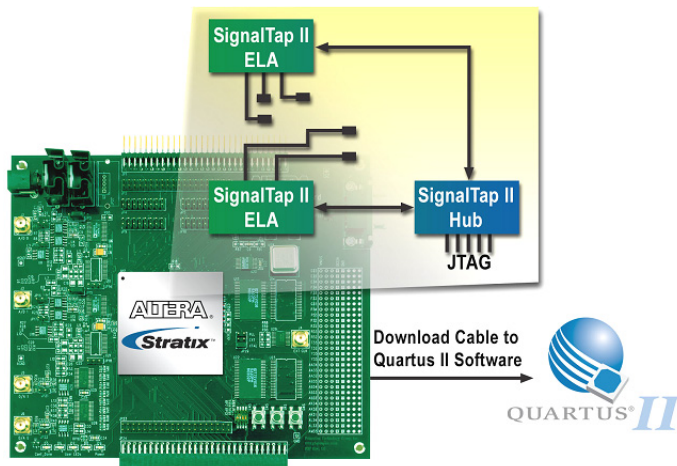
Using the SignalProbe incremental routing feature can significantly reduce the time required for a full recompilation. You can use the SignalProbe incremental routing feature to get quick access to internal design signals to perform system-level debugging.

Introduction

Debugging today's FPGA designs can be a difficult task. As your design continues to increase in complexity, the time and money you invest in verifying your design continues to rise. To get your product to market as quickly as possible, you must minimize the design verification time. To help alleviate the time-to-market pressure, you need a set of verification tools that are powerful and easy to use. The Altera® SignalTap II Logic Analyzer can be used to evaluate the state of the signals in your Altera FPGA, helping you to quickly find the cause of design flaws in your system.

The SignalTap II Logic Analyzer in the Quartus® II software is non-intrusive, scalable, easy to use, and free with your Quartus II subscription. This logic analyzer helps you debug your FPGA design by allowing you to probe the state of the internal signals in your design. It is equipped with many new and innovative features, allowing you to find the source of a design flaw in a short amount of time. Figure 9-1 shows the SignalTap II Logic Analyzer block diagram.

Figure 9-1. SignalTap II Logic Analyzer Block Diagram



This handbook chapter discusses the following topics:

- Including the SignalTap II Logic Analyzer in your design
- Programming the device for SignalTap II analysis
- Advanced features of the SignalTap II Logic Analyzer
- Design examples

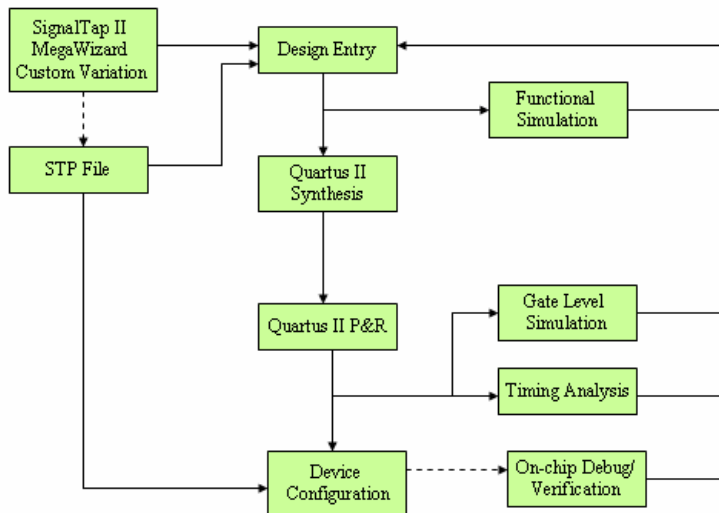
The SignalTap II Logic Analyzer supports the following device families:

- Stratix® II
- Stratix
- Stratix GX
- Cyclone™ II
- Cyclone
- APEX™ II
- APEX 20KE
- APEX 20KC
- APEX 20K
- Excalibur™
- Mercury™

Including the SignalTap II Logic Analyzer in Your Design

There are two ways to build the SignalTap II Logic Analyzer. The first method involves creating a SignalTap II file (.stp) and then defining 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 HDL output module from the MegaWizard in your HDL code.

Figure 9–2 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 custom variation to the final device configuration.

Figure 9–2. SignalTap II Flow

Using the STP File to Create an Embedded Logic Analyzer

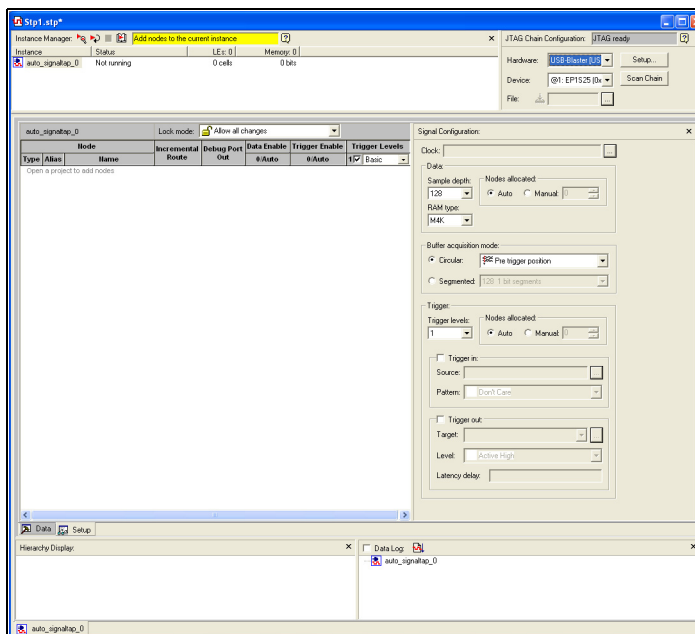
Creating an STP File

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, select **SignalTap II Logic Analyzer** (Tools menu). This method can also be used to create a new STP file.

Both of these methods bring up the SignalTap II window (Figure 9–3).

Figure 9–3. 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. For best results, Altera recommends using a global clock, not a gated clock. Using a gated clock as your acquisition clock, may result in unexpected data that does not accurately reflect your design. The Quartus II Timing Analyzer displays the maximum acquisition clock frequency.

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.
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, select **Start Analysis & Elaboration** (Processing menu). This is particularly useful if you want to quickly add a new node name after you have made design changes.

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**.

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 (zero) 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. The **Trigger Pattern** can be set to any of the following:

- 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 will be used to trigger the analyzer.



For more information on trigger types, see [“Creating Complex Triggers”](#) on page 9–14.

Number of Trigger Levels

The multiple Trigger Level feature gives you precise accuracy 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.

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 have been processed and the final trigger level evaluates to TRUE.

The multiple trigger level feature can be used with Basic Triggers or Advanced Triggers.

You can configure the SignalTap II Logic Analyzer to use up to ten trigger levels. Select the desired number of trigger levels in the **Trigger Levels** list.

You can disable the ability to trigger for a signal by turning off that trigger enable. 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 signal by turning off the data enable column. This option is useful when you want to trigger on a signal, but do not care about 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_signaltap** 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. If you use this method, you do not need to create an STP file and include it in your Quartus II project. 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 file in your HDL, along with using the method described in [“Using the STP File to Create an Embedded Logic Analyzer” on page 9–3](#).

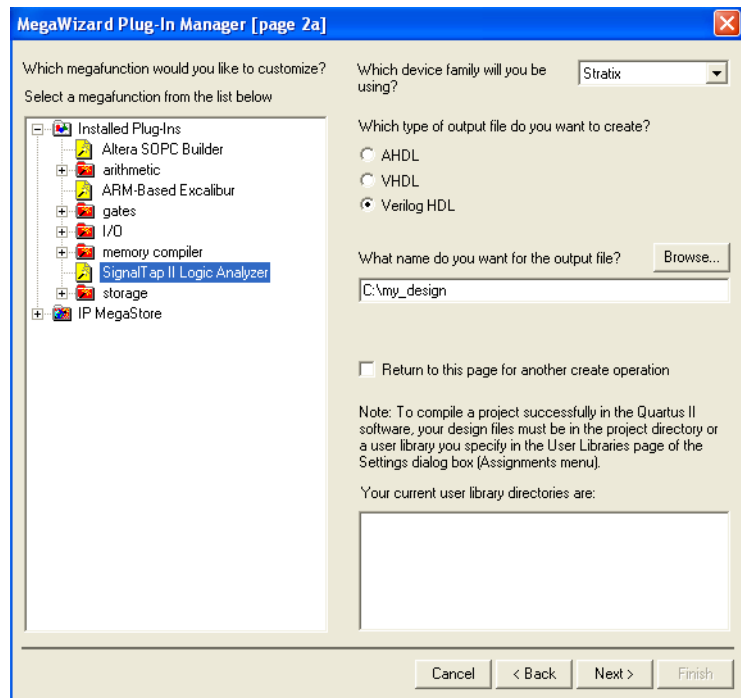
Creating the HDL Representation of the SignalTap II Logic Analyzer

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:

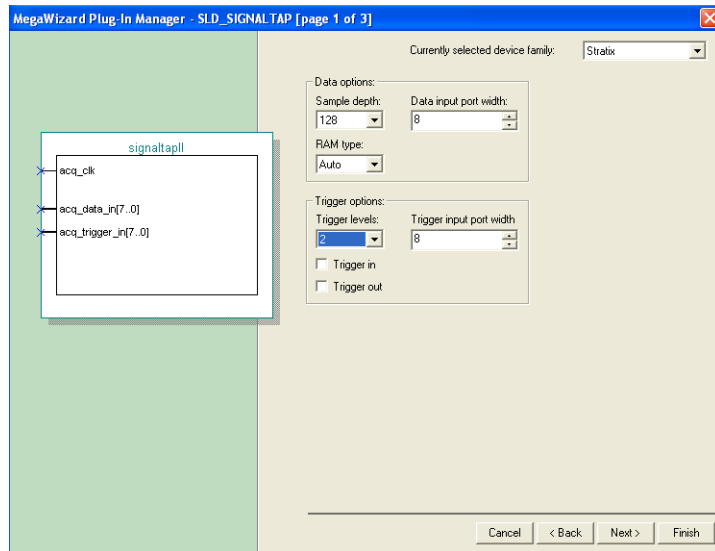
1. Launch the MegaWizard Plug-In Manager by choosing **MegaWizard Plug-In Manager** (Tools menu) in the Quartus II software.

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 9-4](#).

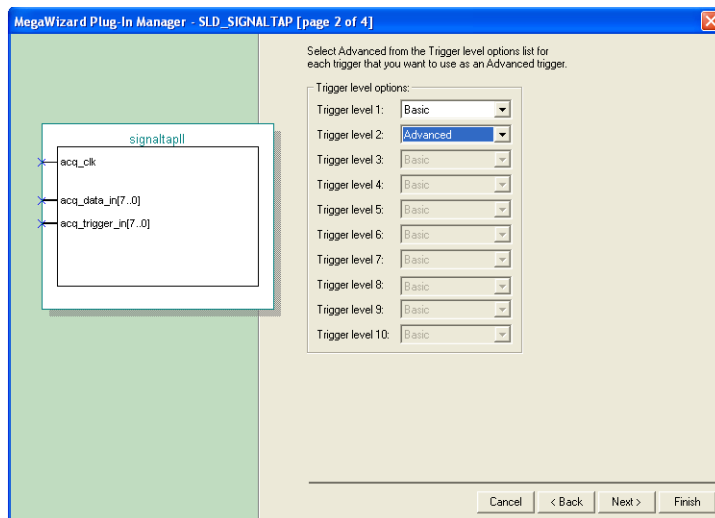
Figure 9-4. Select an Output File and Enter the Selected SignalTap II Name



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 9-5](#).

Figure 9–5. Select the Parameters for the Analyzer

8. Set the Trigger level options by choosing **Basic** or **Advanced**. See [Figure 9–6](#).

Figure 9–6. Basic and Advanced Trigger Options

9. Click **Finish** to complete the process of creating an HDL representation of the SignalTap II Logic Analyzer.

SignalTap II Megafunction Ports

Table 9–1 provides information on the SignalTap II megafunction ports.



Refer to the latest version of the Quartus II software Help for the most current information on the ports and parameters for this megafunction.

Table 9–1. SignalTap II Megafunction Ports			
Port Name	Type	Required	Description
acq_data_in	Input	No	These set of signals represent the signals that are monitored in SignalTap II
acq_trigger_in	Input	No	This set of signals represent the set of signals that are used to trigger the analyzer
acq_clk	Input	Yes	This port represents the sampling clock that SignalTap II uses to capture data
trigger_in	Input	No	This signal is used to trigger SignalTap II
trigger_out	Output	No	This signal is enabled when the trigger event has occurred

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 as many analyzers in your design 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 MegaWizard 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:

1. 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:

- [“Preserving FPGA Memory”](#)
- [“Creating Complex Triggers”](#)
- [“Using External Triggers”](#)
- [“Embedding Multiple Analyzers in One FPGA”](#)
- [“Faster Compilations”](#)
- [“Time Bars and Next Transition”](#)

- “Saving Captured Data”
- “Converting Captured Data to Other File Formats”
- “Creating Mnemonics for Bit Patterns”
- “Capturing Data to a Specific RAM Type”
- “FPGA Resources Used by SignalTap II” II
- “Using SignalTap II in a Lab Environment”
- “Remote Debugging Using SignalTap II”
- “Signal Preservation”
- “Tappable Signals”
- “Timing Preservation with SignalTap II Logic Analyzer”
- “Using SignalTap II Logic Analyzer to Simultaneously Debug Multiple Designs”
- “Locating a Node in the Chip Editor”

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 data-intensive applications in which the amount of saved data exceeds the available sample buffer depth provided by the device RAM. In this signal, 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. Right-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 9–2](#).

Table 9–2. Advanced Triggering Operators <i>Note (1)</i>	
Name of Operator	Type
Less Than	Comparison
Less Than or Equal To	Comparison
Equality	Comparison
Inequality	Comparison
Greater Than	Comparison
Greater Than or Equal To	Comparison
Logical NOT	Logical
Logical AND	Logical
Logical OR	Logical
Logical XOR	Logical
Reduction AND	Reduction
Reduction OR	Reduction
Reduction XOR	Reduction
Left Shift	Shift
Right Shift	Shift
Bitwise Complement	Bitwise
Bitwise AND	Bitwise
Bitwise OR	Bitwise
Bitwise XOR	Bitwise
Edge and Level Detector	Signal Detection

Note to Table 9–2:

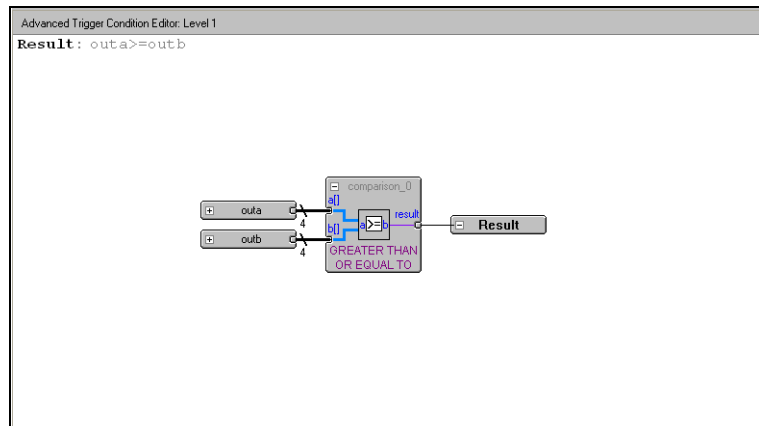
(1) For more information on each of these operators, see Quartus II Help.

Some of the operators have the ability to be configured 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.

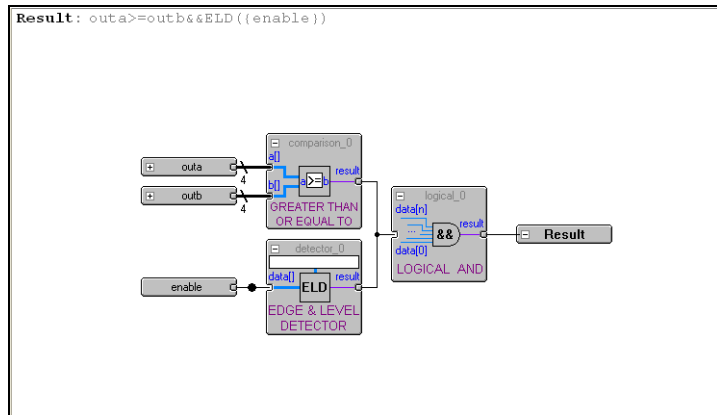
The following examples show how to use Advanced Triggering:

- Trigger when bus `outa` is greater than or equal to `outb` (see [Figure 9-7](#))

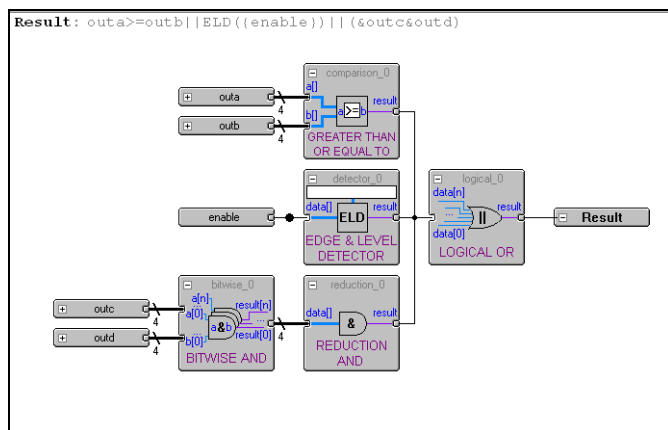
Figure 9-7. Bus Out a is Greater Than or Equal to Out b



- Trigger when bus `outa` is greater than or equal to `outb`, and when the enable signal has a rising edge (see [Figure 9-8](#))

Figure 9–8. Enable Signal Has a Rising Edge

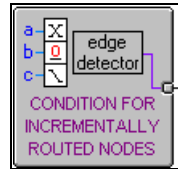
- Trigger when bus `outa` is greater than or equal to bus `outb`, or when the enable signal has a rising edge. Or, when a bitwise AND operation has been performed with bus `outc` and bus `outa`, and all bits of the result of that operation are 0 (see Figure 9–9).

Figure 9–9. Bitwise AND Operation

The advanced triggering capability can only be used with pre-synthesis nodes. Post-fitting nodes can only be used for basic trigger operations. 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 (as shown in Figure 9–10) appears in the advanced trigger panel.

Figure 9–10. Symbol for STP File Containing Post-Fitting Nodes



The output of this symbol can be combined with the operators listed in Table 9–2.

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 be operated 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 the **Trigger In**.

When the **Node Finder** window appears, select an input pin in your design by setting the **Trigger In** source.

Trigger Out

To use Trigger Out, perform the following steps:

1. In the SignalTap II window, click the **Setup** tab.

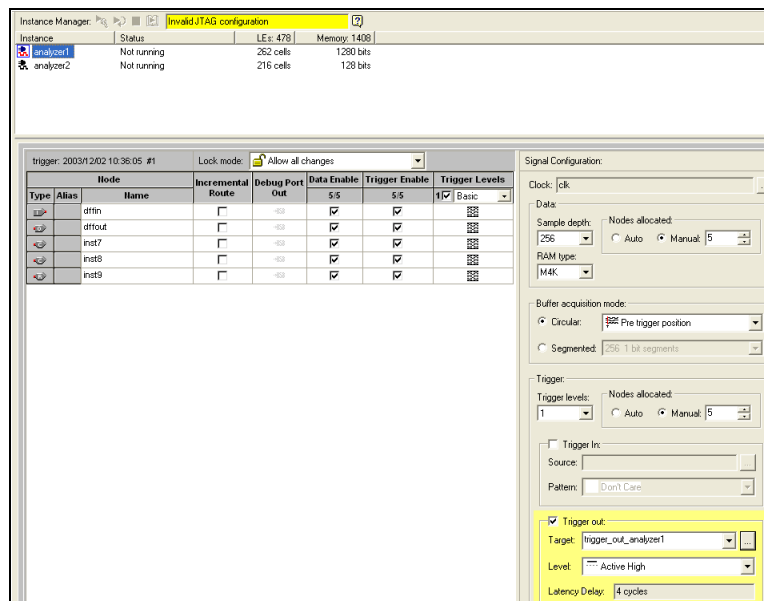
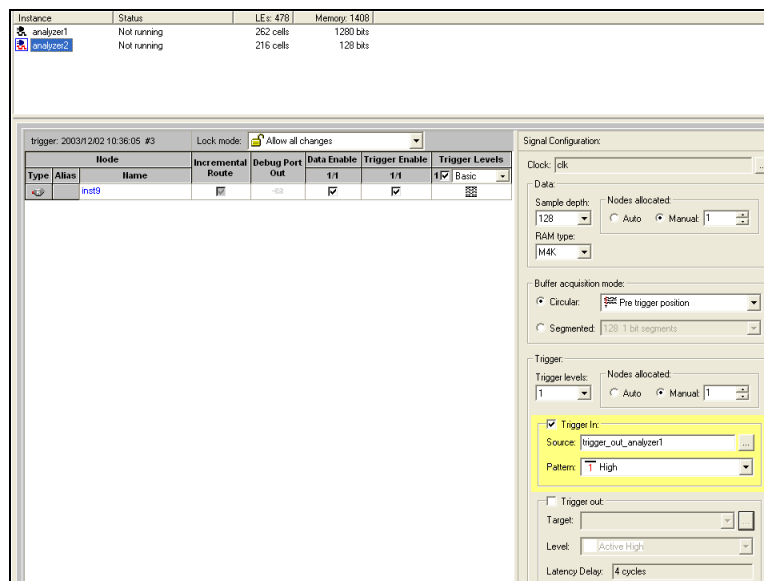
2. In the **Signal Configuration** window, turn on **Trigger Out**.
3. In the **Level** list, select the condition you would like to signify that the trigger event is occurring.
4. Click Browse next to the **Trigger Out**.

When the **Node Finder** window appears, select an output pin in your design.

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 9-11](#)). 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 9-12](#)).

Figure 9–11. Enabling the Trigger Out Signal**Figure 9–12. 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 clicking **Run** on the SignalTap II toolbar.

Faster Compilations

The incremental routing feature allows you to add new nodes to your STP file without having to perform a full recompilation. Adding these new nodes to your STP file does not affect the existing placement and routing of your design. Before using the SignalTap II incremental routing feature, you must perform the following steps:

1. Set the number of nodes allocated.
2. Select any nodes reserved for incremental routing.

Set the Number of Nodes Allocated

Before you can fully utilize the incremental routing feature you must first select **Manual** under **Nodes allocated**, as shown in [Figure 9–13](#), and enter a value that includes the number of nodes you currently want to analyze, plus any extra nodes you may want to incrementally route later in the verification process. The extra allocated nodes act as place-holders for nodes that you will add later.

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 9–13. Nodes Allocated

Data:

Sample depth: 128

RAM type: M4K

Nodes allocated: ☐ Auto ☒ Manual: 20

Select Nodes Reserved for Incremental Routing

As shown in [Figure 9–14](#), 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 9–14. The SignalTap II Setup Window *Note (1)*

Node			Incremental Route	Debug Port Out	Data Enable	Trigger Enable	Trigger Levels
Type	Alias	Name			68/Auto	68/Auto	1/Advanced
		inf_b			<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"/>	
		a[0]			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	

Note to Figure 9–14:

- (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 are running into a situation where the Quartus II software is not able to route your signal

you can 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 and 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 a signal, you can use either the **Next Transition** or the **Previous Transition** button.

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)

To export SignalTap II Logic Analyzer's captured data, choose **Export** (File menu) and select the **File Name**, the **Export Format**, and the **Clock Period**.

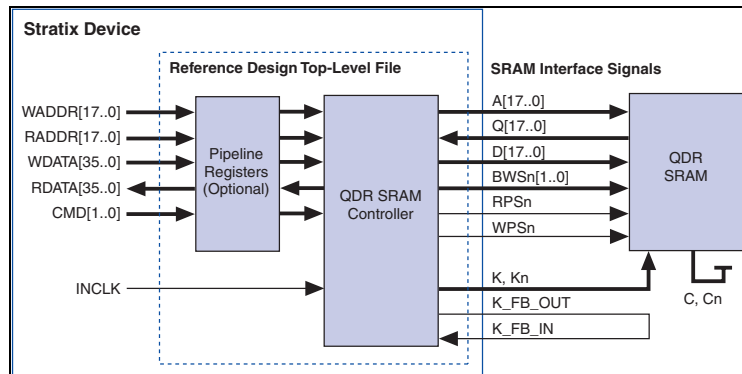
Creating Mnemonics for Bit Patterns

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**.

Buffer Acquisition

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 9–15](#).

Figure 9–15. Example System Generating Periodic Events



SignalTap II Logic Analyzer can be used to verify functionality of the design shown in [Figure 9–15](#) and ensure that the correct data are written to the SDRAM controller. The buffer acquisition in SignalTap II Logic Analyzer allows you to monitor the RDATA port when H' 0F0F0F0F is sent into the RDADDR port. You have the ability to monitor multiple read transactions from the SDRAM device without re-running 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 varies depending 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

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 tables provides an estimate of the number of LEs and the amount of memory that are required to add SignalTap II Logic Analyzer to your design.

Table 9–3 shows the SignalTap II Logic Analyzer M4K memory block resource usage for these devices per signal width and sample depth.

Table 9–3. SignalTap II Logic Analyzer M4K Block Utilization for Cyclone, Stratix GX, and 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 9–3:

- (1) When configuring a SignalTap II 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 of SignalTap II 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.

Another useful feature that is part of the SignalTap II interface in the Quartus II software is the SRAM Object File (SOF) Manager. This feature allows you to archive multiple SOFs that have different SignalTap II configurations into one STP file. For more information on how to use this feature refer to the Quartus II help.

Remote Debugging Using SignalTap II

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:

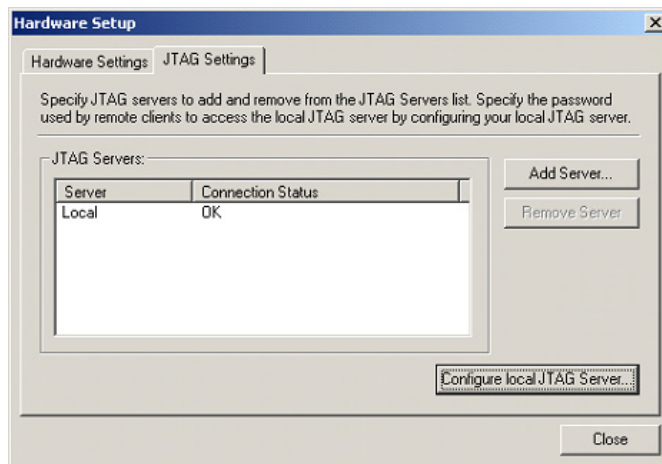
- Quartus II software installed on the local PC
- Stand-alone SignalTap II installed on the remote PC
- Programming hardware connected to the PCB at the remote location
- TCP/IP connection

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 a connected Altera programming hardware, for example, USB-Blaster™ or ByteBlaster™.
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 - Remote PC:

1. Select **Hardware Setup** from the Quartus II programmer.
2. Select the **JTAG Settings** tab. See [Figure 9–16](#).
3. Click the **Configure local JTAG server** button.

Figure 9–16. Configure Hardware Settings

4. In the **Configure Local JTAG Server** dialog box (see [Figure 9–17](#)), 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**.

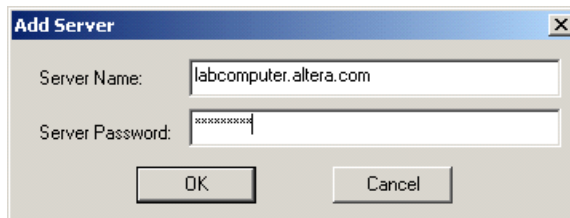
Figure 9–17. Configure Local JTAG Server



Software Setup - 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 9–18](#)), 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.

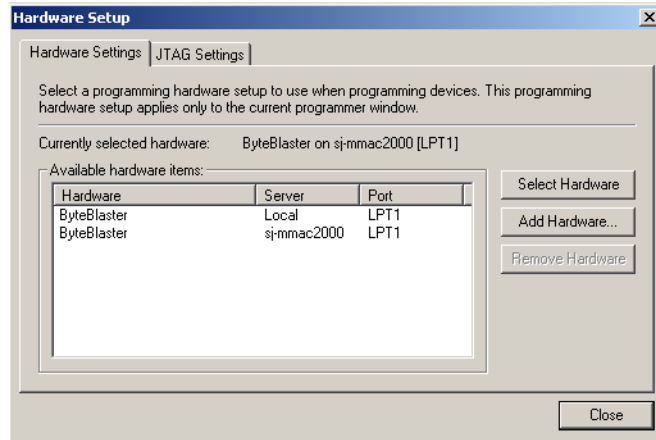
Figure 9–18. Add Server Dialog Box



5. Click **OK**.

SignalTap II Setup - Local PC

1. Select the hardware by clicking the **Hardware Setup** tab and choosing the hardware on the Remote PC. See [Figure 9–19](#).

Figure 9–19. 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 may be optimized during the process of synthesis and place-and-route. This may lead to issues when you are attempting to debug your design, because the post-fitting signal names differ significantly from your RTL names. To avoid this issue you may need to use the 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 may see an increase in resource utilization and/or a decrease in timing performance. The two attributes you may be able to use are:

- **Keep**—This attribute ensures that combinational signals do not get removed.
- **Preserve**—This attribute ensures that registers do not get 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. The types of signals that cannot be tapped are listed below:

- 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 of a SERDES block.

Timing Preservation with SignalTap II Logic Analyzer

In addition to verifying functionality, timing closure is one the most crucial processes in successfully completing a design. When you compile a project with SignalTap II Logic Analyzer, you are adding IP to your existing design, therefore you could potentially affect the existing placement and routing, and the timing of your design. To minimize the effect that 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.

For an example of timing preservation with SignalTap II, 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: a Quartus II project does not have to be open 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.

Figures 9–20 through 9–23 show a JTAG chain and its associated STP files.

Figure 9–20. JTAG Chain

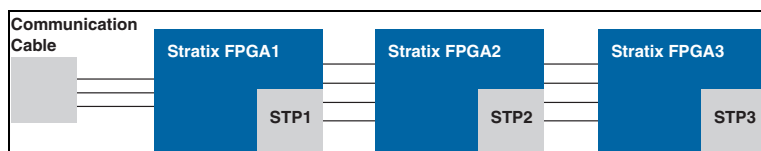


Figure 9–21. STP File for the First Device in the JTAG Chain

Instance Manager: Ready to acquire

Instance	Status	LEs: 281	Memory: 640
auto_signaltap_0	Not running	281 cells	640 bits

JTAG Chain Configuration: JTAG ready

Hardware: USB-Blaster [US] Setup...

Device: 1: EP1525 [0x] Scan Chain

File: top_level.sof

trigger: 2003/12/18 16:59:59 #1 Lock mode: Allow all changes

Node			Incremental Route	Debug Port Out	Data Enable	Trigger Enable	Trigger Levels
Type	Alias	Name			5/Auto	5/Auto	1/Basic
		clear	<input type="checkbox"/>		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
		crit_enable	<input type="checkbox"/>		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
		digit	<input type="checkbox"/>		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	XXXb

Signal Configuration:

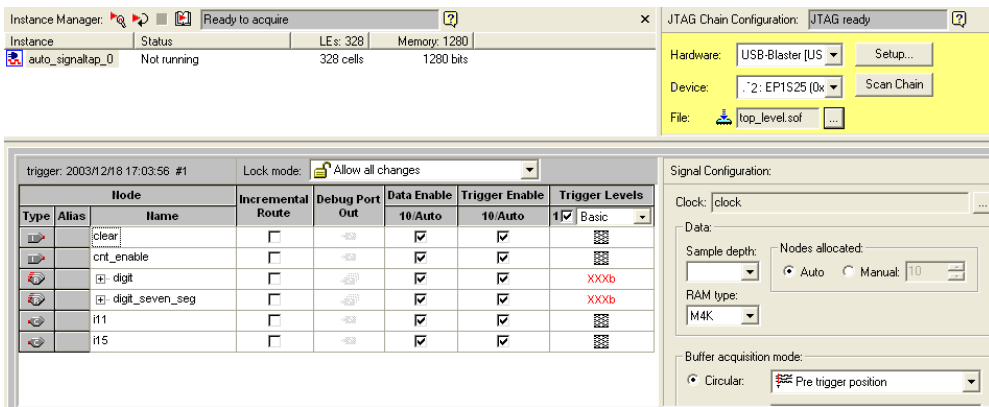
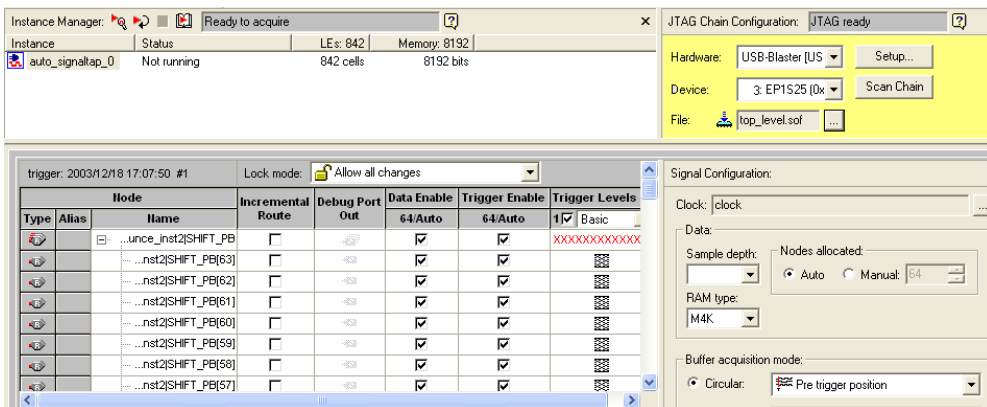
Clock: clock

Data:

Sample depth: Nodes allocated: ☒ Auto ☐ Manual:

RAM type: M4K

Buffer acquisition mode: ☒ Circular ☐ Pre trigger position

Figure 9–22. STP File for the Second Device in the JTAG Chain**Figure 9–23. STP File for the Third Device in the JTAG Chain**

Locating a Node in the Chip Editor

Once you have found the source of a bug in your design using SignalTap II Logic Analyzer, you can locate that signal in the Chip Editor. Then, you can change your design to correct the flaw that was found using 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*.

Design Example: Preserving Timing

The following example shows the importance of back annotating your design prior to inserting 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.

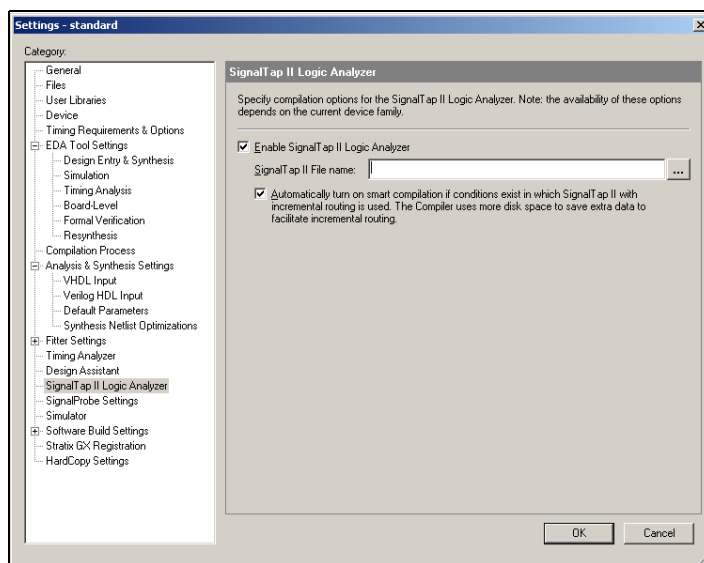
1. Initial compilation without SignalTap II Logic Analyzer

When you run the Quartus II Timing Analyzer, you see the following results for the main clock in the design (see [Table 9–4](#)).

Table 9–4. 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 SignalTap II Logic Analyzer

You are debugging this design with SignalTap II Logic Analyzer, so you must compile it with an STP file. To enable 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 9–24](#).

Figure 9–24. Enabling the STP File in the SignalTap II Logic Analyzer Page

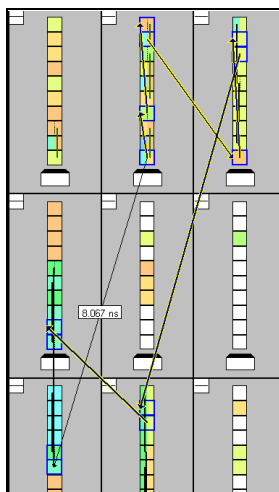
Once the compilation is complete, the results shown in [Table 9–5](#) are reported by the Quartus II Timing Analyzer.

Table 9–5. 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 9–25](#) shows a failing path in the timing closure floorplan editor.

Figure 9–25. Failing Path in the Timing Closure Floorplan Editor

3. Back-annotate the original design

To minimize the effect that SignalTap II Logic Analyzer has on the original design, you should back-annotate the design to constrain it to a portion of the FPGA. This is done by selecting **Back-Annotate Assignments** (Assignments menu). After you have back-annotated your design, it is safe to insert SignalTap II Logic Analyzer to your project.

Compile the design and you will see the results shown in [Table 9–6](#).

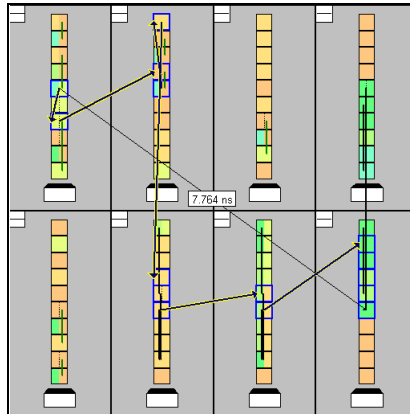
Table 9–6. F_{MAX} Results from the Quartus II Timing Analysis with SignalTap II Logic Analyzer After Back-Annotation

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 decreased significantly and the original timing requirements have been met.

Figure 9–26 shows the timing closure floorplan editor.

Figure 9–26. 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 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 must face is implementing incremental engineering change orders (ECOs) late in the design cycle while maintaining timing closure. With the Quartus® II software's new Chip Editor, you can view the internal structure of Altera® devices and incrementally edit device resource functionality and parameter settings. The Chip Editor can also help you document and manage ECOs.

The Chip Editor works directly on design netlists so you can implement device changes in minutes without performing a design compilation. Changes are restricted to a particular device resource to maintain timing closure in the remaining portions of the design. Design rule checks are performed on all changes to prevent you from making illegal edits.

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:

- FPGA routing resources used by your design. For example, you can visually examine how blocks are physically connected, as well as the signal routing that connects the blocks.
- LE utilization information: 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 or the look-up table (LUT) or both, as well as the signal flow through the LE.
- ALM utilization information: 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 utilization information: You can view how the device I/O resources are used. For example, you can view what components of the I/O are used, if the delay chain settings are enabled, and the signal flow through the I/O.

- PLL utilization information: 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 the following elements within the Altera device:

- Logic elements
- I/O cells
- Phase-locked loops (PLL)



With the Chip Editor, you can view the contents of an ALM and its implementation, but you cannot edit its properties.



For more information on the Change Manager, see [“Change Manager” on page 10–23](#).

The Chip Editor can be used with the following device families:

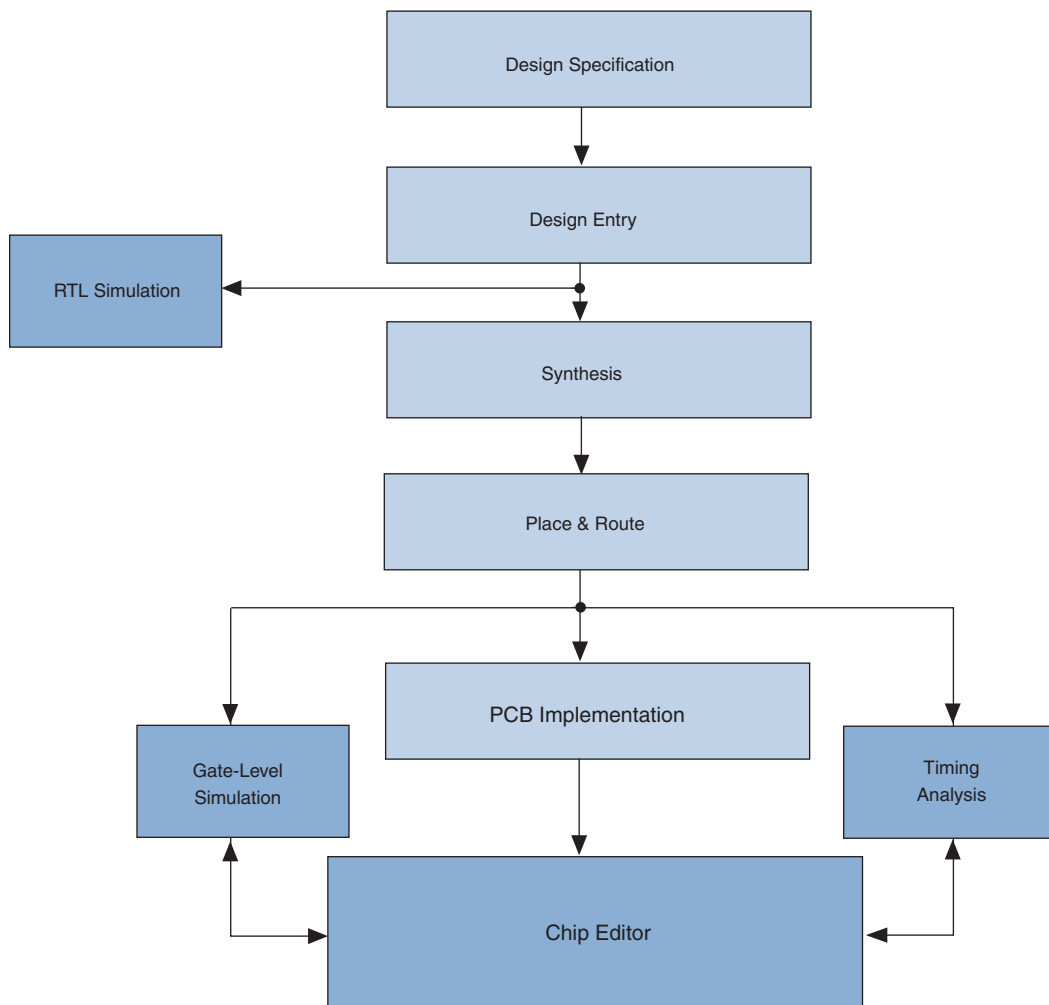
- Stratix® II
- Stratix
- Stratix GX
- Cyclone™
- MAX® II

Using the Chip Editor in Your Design Flow

An 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 worse—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.

With the Altera Chip Editor, you can significantly shorten the design cycle time, and ultimately the time to market for your product. You can make changes directly to the post place-and-route netlist, generate a new programming file, and test the revised design without ever modifying the RTL code. [Figure 10–1](#) describes how the Chip Editor can be used in your design flow.

Figure 10–1. Chip Editor Design Flow

Chip Editor Overview

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 examine FPGA resources used by your design
- Resource Property Editor: Allows you make modifications to your post place-and-route design
- Change Manager: Allows you to track all design changes

Chip Editor Floorplan

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**

The Chip Editor 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.

Table 10–1 gives a summary of the Chip Editor features. These features are easily accessible from the Chip Editor Toolbar.

Table 10–1. Chip Editor Floorplan Features	
Feature	Description
Birds Eye View	Gives a high-level picture of resource usage at the chip level, allows you to specify which elements of the Chip Editor are displayed, and assists you in rapidly navigating the floorplan
Fan-In Connections	Displays the connections to the selected resource
Fan-Out Connections	Displays the connections away from the selected resource
Immediate Fan-In	Highlights the resource that directly feeds the selected element
Immediate Fan-Out	Highlights the resource that is directly fed by the selected element
Show Delays	Displays the time delay between the two selected resources

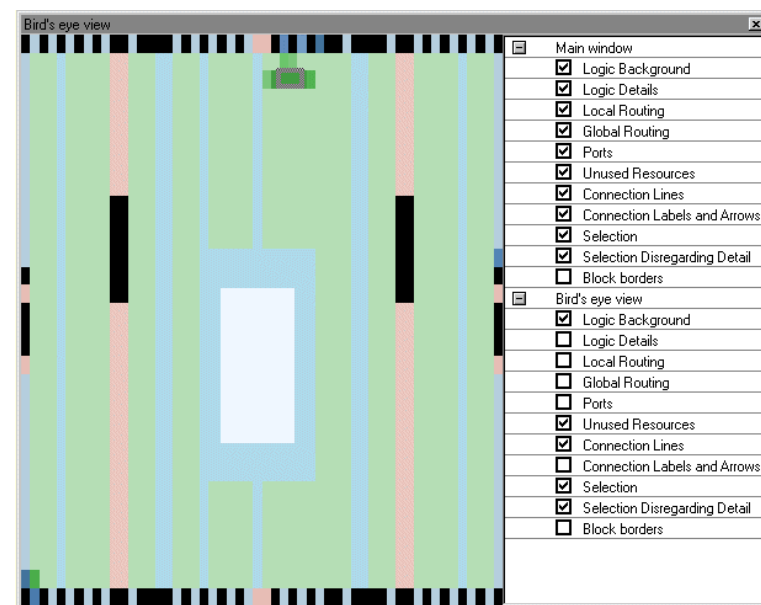


For more information on the Chip Editor Toolbar refer to the Quartus II on-line help.

Bird's Eye View

The Bird's Eye View (see [Figure 10-2](#)) 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 controls apply to both the Bird's Eye View and the main Chip Editor window.

Figure 10-2. Bird's Eye View



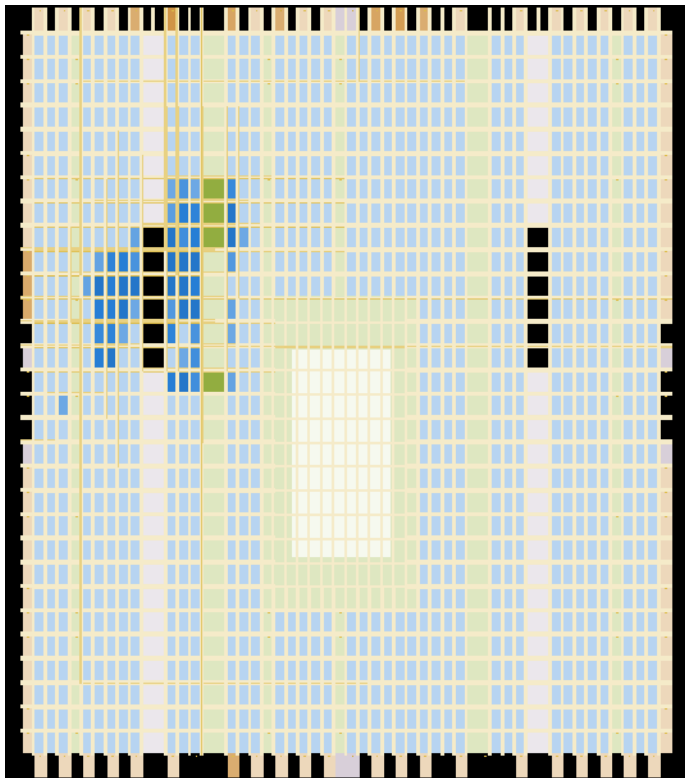
The Bird's Eye View is displayed as a separate window that is linked to the Chip Editor. When you select an area of interest in the Bird's Eye View, the Chip Editor automatically refreshes the window as necessary to display the selected area in greater detail, in accordance with whatever zoom factor is in effect. For example, when you zoom-in (or zoom-out) in the Bird's Eye View window, the main Chip Editor window will also zoom-in (or zoom-out). You have the option of setting the amount of detail that you see when you use the zoom-in feature. To adjust the default values, specify the appropriate values on the Chip Editor page of the **Options** dialog box (Tools menu).

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.

First (Highest) Level View

The first (highest) zoom level provides a high-level view of the entire device floorplan. This view provides a similar level of detail as the Quartus II Timing Closure floorplan. You can easily locate and view the placement of any node in your design. [Figure 10–3](#) shows the Chip Editor's first level view.

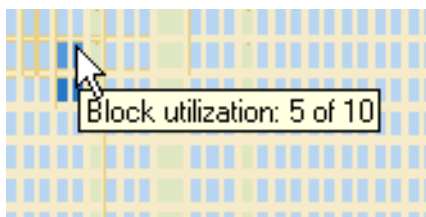
Figure 10–3. Chip Editor's First (Highest) Level View



Each resource is shown in a different color, making it easier to distinguish between resources. The Chip Editor uses a gradient color scheme: the color becomes darker as the utilization of a resources 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 10-4](#)).

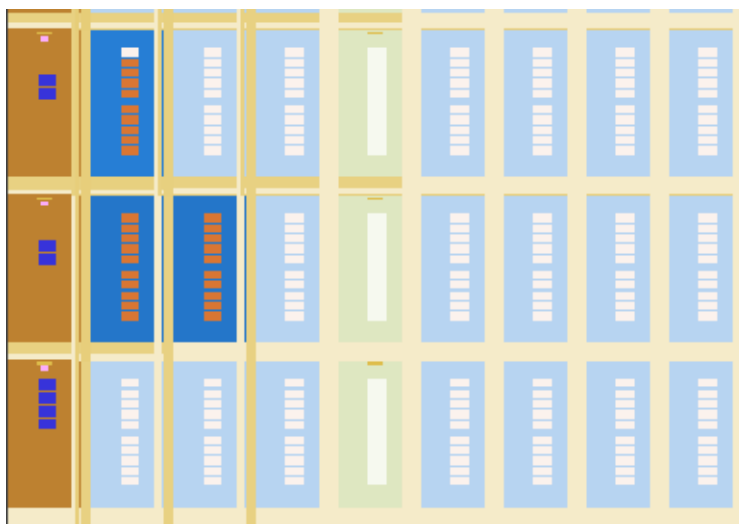
Figure 10-4. Tooltip Message: First Level View



Second Level View

As you continue to zoom in, you see an increase in the level of detail. [Figure 10-5](#) shows the Chip Editor's second level view.

Figure 10-5. Chip Editor's Second Level View

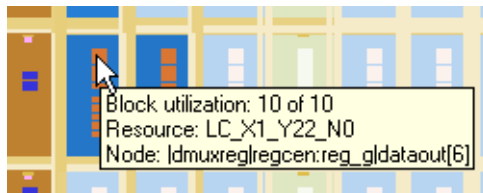


At this level you can see the contents of LABs and I/O banks. You also see the routing channels that are used to connect resources together.

When you place the mouse pointer over an LE at this level, a tooltip is displayed that describes 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 10–6 shows the level 2 tooltip information.

Figure 10–6. Tooltip Message: Second Level View

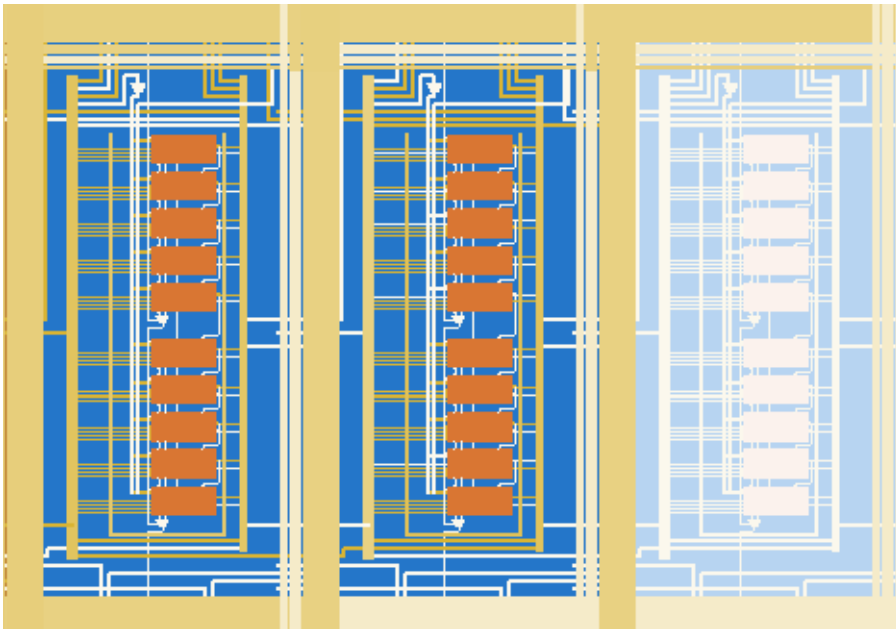


Third Level View

Figure 10–7 shows the level of detail at the third and lowest level. At this level you can see within the FPGA. You can see each routing resource that is used with a LAB.

You also have the ability to 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 have the ability to create new LEs and I/Os. To create a resource, right-mouse click at the location you want to create the resource and select **Create Atom**.

Figure 10–7. Chip Editor's Third Level View



Resource Property Editor

You can view the following elements with the Resource Property Editor:

- LEs
- ALMs
- I/O elements
- PLLs

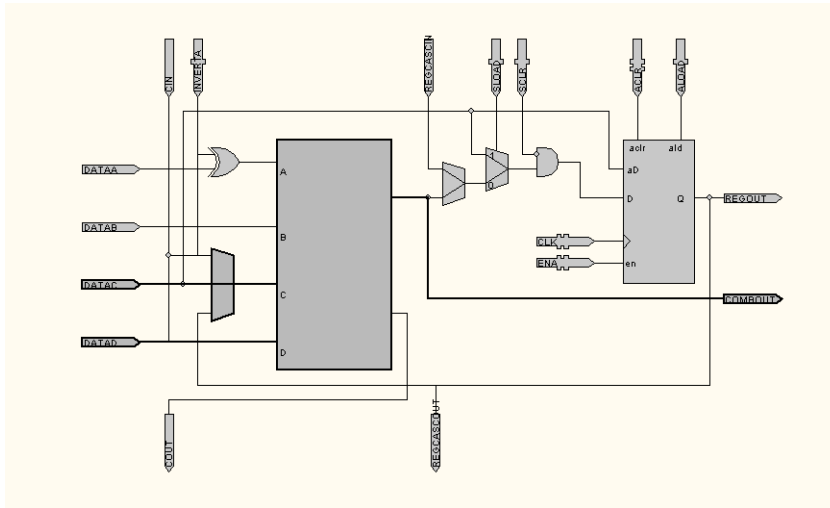
The Logic Element (LE)

An Altera logic element 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 10–8](#) shows what the LE looks like in the Resource Property Editor.

Any LE that is placed in the FPGA can be viewed and edited using the Resource Property Editor. To launch the Resource Property Editor for an LE, right-mouse click on an LE in the Timing Closure Floorplan, Last Compilation Floorplan, Node Finder, or Chip Editor and select **Locate in Resource Property Editor** from the menu.

For a detailed description of the LE for a particular device family, refer to the Handbook or data sheet for the device family.

Figure 10–8. Stratix LE Architecture



The Adaptive Logic Module (ALM)

The basic building block of logic in the Stratix II architecture is the Adaptive Logic Module (see [Figure 10-9](#)). 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 also 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 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, Last Compilation Floorplan, or Node Finder, and select **Locate in Resource Property Editor** from the right button pop-up menu.

Figure 10–9. ALM Architecture

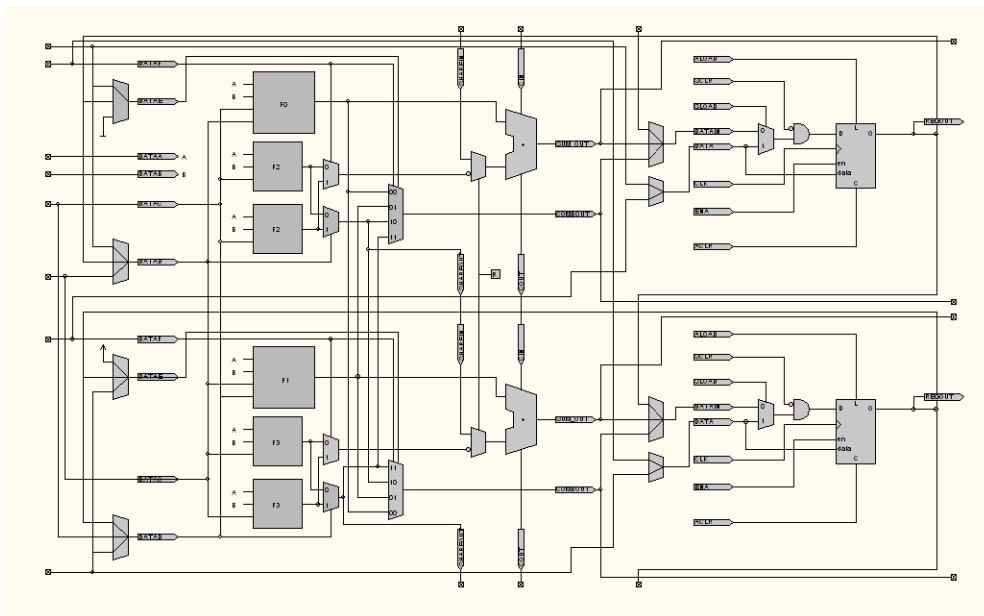


Table 10-2 shows which operations are supported for various device families.

Operation	Stratix	Stratix GX	Stratix II	Cyclone	MAX II
View the LEs/ALMs in the Resource Property Editor	✓	✓	✓	✓	✓
Edit properties of the LEs/ALMs	✓	✓		✓	✓
Placement changes the LEs/ALMs	✓	✓		✓	✓
Create new LEs/ALMs	✓	✓		✓	✓

Properties of the Logic Element

This section discusses the following properties of the logic element that can be examined using the Resource Property Editor:

- Mode of operation
- LUT equation
- LUT mask
- Synchronous mode
- Register cascade mode

Mode of Operation

An LE can operate in either normal or arithmetic mode. For more information on 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 configured in normal mode, the LUT can implement a function of four inputs.

When configured in arithmetic mode, the LUT is 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

The LUT equation allows you to change the logic equation that is currently implemented by the LUT. 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.

When a change is made to the LUT equation, the Quartus II software automatically changes the LUT mask.

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, and Cyclone 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.

Assume that you need to build the following logic function:

$$(A \text{ XOR } B) \text{ OR } (C \text{ AND } D)$$

LUT Mask

To generate the LUT mask, the truth table for an equation must be computed. Table 10–3 lists the truth table for logic equation from the section above:

<i>Table 10–3. LUT Mask Truth Table</i>				
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 $(A \text{ XOR } B) \text{ OR } (C \text{ AND } D)$ is represented by the following binary number: 1111011001100110. The LUT mask, in hexadecimal, 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 cannot remove VCC connections to the `sload`, however if you want to change the synchronous mode of the LE to off, you can connect the `sload` and `sclr` to a valid GND signal in your design.

You can invert either the `sload` or `sclr` signal feeding into the LE. The `sload` signal, if used in an LE, 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.

Properties of an ALM

LUT Mask

As mentioned in the section above, 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 the above example. 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`, 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, you will get the following:

LUT Mask: 0000FFFFFFFF0000

COMBOUT Equation: (!DATAE & DATAF) # (!DATAF & DATAE)

Extended LUT Mode

When the extended LUT mode is used, the ALM creates a specific set of seven-input functions. The Quartus II software automatically recognizes the applicable 7-input function and fits them into an ALM. The 'top' and 'bottom' LUTs are both a function of five inputs, where four of the inputs are shared. The other input of the ALM is used to control the 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

Stratix, Stratix GX, and 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 10–10 shows the Stratix I/O element structure. The I/O element contains two input registers (plus a latch), two output registers, and two output enable registers.

Figure 10–10. Stratix Device I/O Element

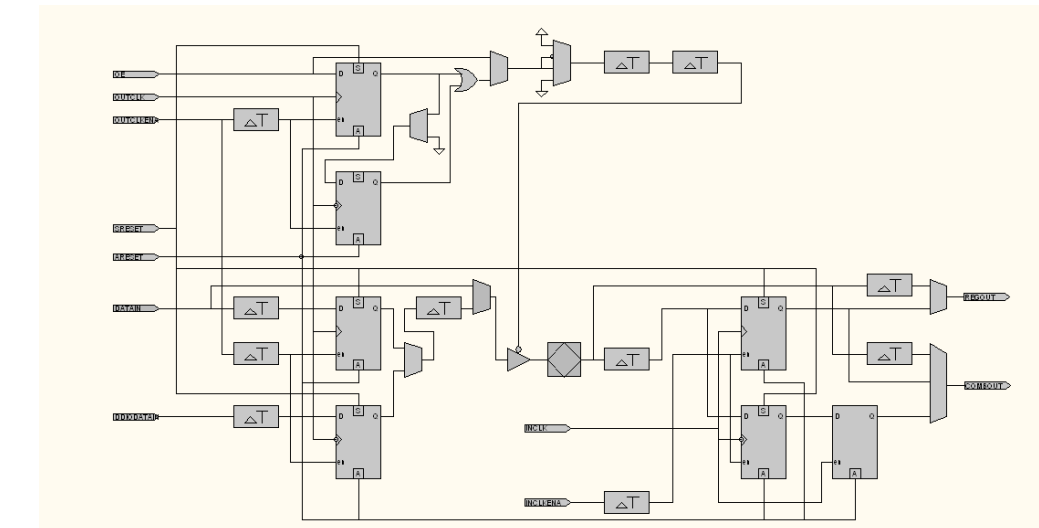
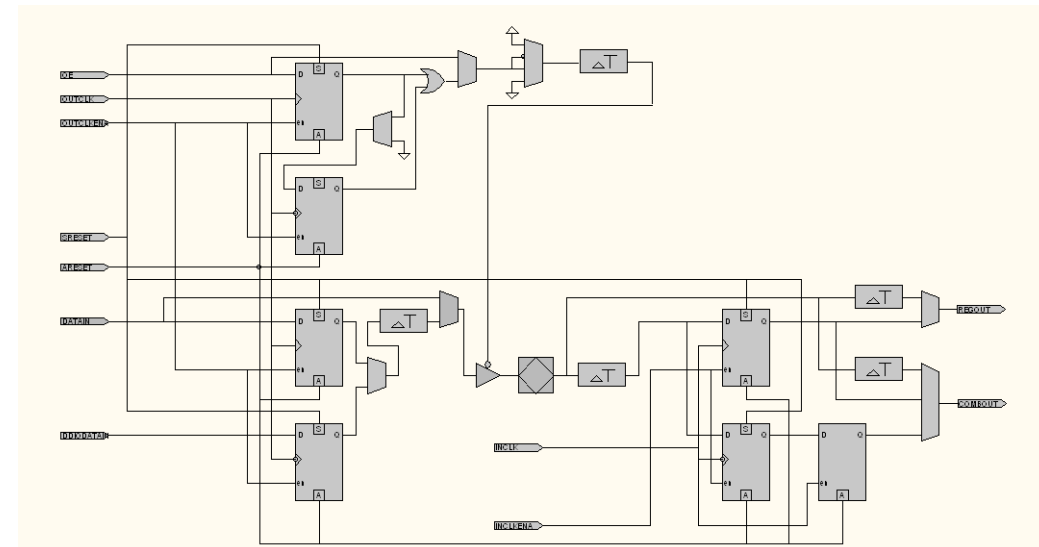


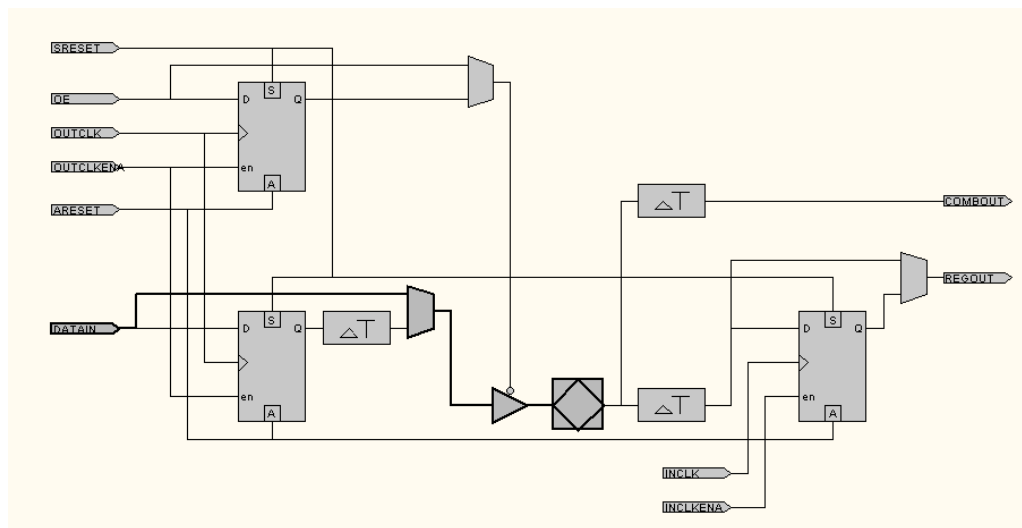
Figure 10–11 shows the Stratix II I/O element structure.

Figure 10–11. Stratix II Device I/O Element



The I/O element in Cyclone device contain a bidirectional I/O buffer and three registers for complete bidirectional single data rate transfer.

Figure 10–12. Cyclone Device I/O Element



MAX II device I/O elements contain a bidirectional I/O buffer.

Altera Corporation
June 2004

Editable Properties of I/O Elements

Stratix and Stratix GX Properties

You can use the Resource Property Editor to modify the following properties of Stratix and Stratix GX device I/O cells:

- Bus Hold
- Weak Pull Up
- Slow Slew Rate
- Open Drain
- I/O Standard
- Current Strength
- Extend OE Disable
- PCI I/O
- On-Chip Termination
- Input Register Mode
- 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 t_{CO} Delay
- Output t_{ZX} Delay
- Logic Array to Output Register Delay

Stratix II Properties

You can use the Resource Property Editor to view the following properties of Stratix II device I/O cells:

- Bus Hold
- Weak Pull Up
- Slow Slew Rate
- Open Drain

- I/O Standard
- Current Strength
- Extend OE Disable
- PCI I/O
- On-Chip Termination
- Input Register Mode
- 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
- Output Enable Clock Enable Delay
- Input Pin to Logic Array Delay
- Output Pin Delay
- Input Pin to Input Register Delay
- Output Enable Register t_{CO} Delay

Cyclone Properties

You can use the Resource Property Editor to modify the following properties of Cyclone device I/O cells:

- Bus Hold
- Weak Pull Up
- Slow Slew Rate
- Open Drain
- I/O Standard
- Current Strength
- Extend OE Disable
- PCI I/O
- On-Chip Termination
- Input Register Mode
- 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 Pin to Logic Array Delay
- Output Pin Delay
- Input Pin to Input Register Delay

Max II Properties

You can use the Resource Property Editor to modify the following properties of MAX II device I/O cells:

- Bus Hold
- Weak Pull Up
- Slow Slew Rate
- Open Drain
- I/O Standard
- Current Strength
- Extend OE Disable
- PCI I/O
- Input Pin to Logic Array 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.

Properties of the PLL

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 VCO Tap
- 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:

$$\begin{aligned}\text{High \%} &= \text{Counter High} / (\text{Counter High} + \text{Counter Low}) \\ \text{Low \%} &= \text{Counter Low} / (\text{Counter High} + \text{Counter Low})\end{aligned}$$

Adjusting the Phase Shift

Use the following equations to adjust the phase shift of an output clock of a PLL:

$$\text{Phase Shift} = (\text{VCO Period} * 1/8 * \text{VCO Tap}) + (\text{VCO Init} * \text{VCO Period})$$

Normal Mode

$$\begin{aligned}\text{VCO Tap} &= \text{Counter PH} - \text{M VCO Tap} \\ \text{VCO Init} &= \text{Counter Initial} - \text{M Initial} \\ \text{VCO Period} &= \text{In Clock Period} * \text{N} / \text{M}\end{aligned}$$

External Feedback Mode

$$\begin{aligned}\text{VCO Tap} &= \text{Counter PH} - \text{M VCO Tap} \\ \text{VCO Init} &= \text{Counter Initial} - \text{M Initial} \\ \text{VCO Period} &= \text{In Clock Period} * \text{N} / (\text{M} + \text{Counter High} + \text{Counter Low})\end{aligned}$$

Adjusting the Output Clock Frequency

Use the following equations to adjust the output clock of a PLL.

Normal Mode

$$\text{OUTCLK} = \text{INCLK} ((\text{M})/(\text{N})(\text{Counter High} + \text{Counter Low}))$$

External Feedback Mode

$$\text{OUTCLK} = \text{INCLK} ((M + \text{Counter High} + \text{Counter Low}) / (N)(\text{Counter High} + \text{Counter Low}))$$

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:

$$\% \text{spread} = 1 - \frac{M_2 N_1}{M_1 N_2}$$



For a detailed description of the settings, see Quartus II Help. For more information on Stratix device PLLs, see the *Stratix Architecture* chapter in Volume 1 of the *Stratix Device Handbook*.

Change Manager

The Change Manager allows you to track all the design changes made with the Chip Editor. [Table 10–5](#) summarizes the information shown by the Change Manager.

Table 10–5. 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)
Status	Current state of the change made to the node specified
Comments	User comments

The current state of your change can be viewed in the Change Manager. When the **Check & Save All Netlist Changes** function is performed, you will see the status of the change in the Change Manager. See [Figure 10–14](#).

Figure 10–14. Change Manager Results

	Node Name	Change Type	Old Value	Target Value	Current Value	Disk Value	Status
1	test out2	Location Index	IOC_X52_Y...	IOC_X52_Y...	IOC_X52_Y...	IOC_X52...	Committed
2	test out2	Current Strength	24mA	16mA	16mA	16mA	Committed
3	test in2	Location Index	IOC_X52_Y...	IOC_X52_Y...	IOC_X52_Y...	IOC_X52...	Committed
4	test in1	Location Index	IOC_X52_Y...	IOC_X52_Y...	IOC_X52_Y...	IOC_X52...	Applied
5							

Table 10–6 describes the values that appear in the Status column of the Change Manager.

Table 10–6. Status Values in the Change Manager	
Value	Description
Applied	A change has been made and saved, but Check & Save All Netlist Changes has not been performed
Committed	A change has been made, saved, and Check & Save All Netlist Changes has been performed
Not Valid	A change has been made and saved. A new change to the same element that supersedes the original change results in the status being set to “Not Valid”.
Not Applied	A change has been made and saved. However, if the original value has been restored, the newly created entry appears as “Not Applied”.

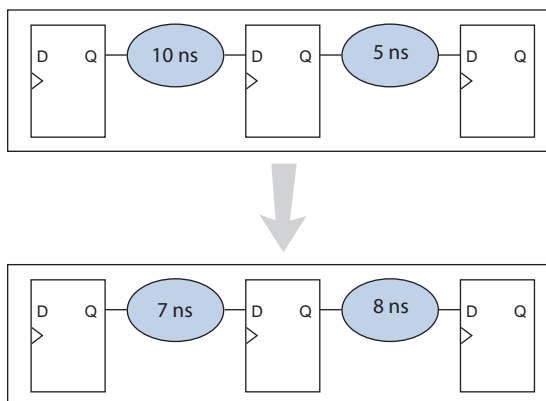
Common Applications

The Chip Editor can be used in a number of ways to help build your system as quickly as possible. The list below shows some of the ways you can use the Chip Editor:

- Gate-level register retiming
- 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

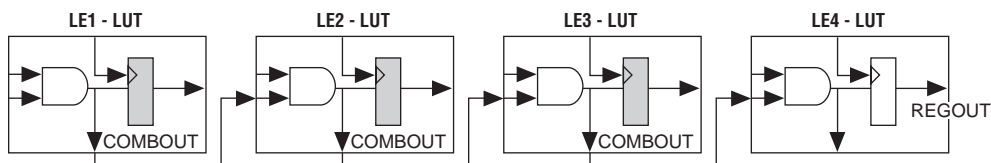
Gate-Level Register Retiming

Retiming your design involves moving registers to balance the combinational delay across a data path, while preserving the overall functionality of the circuit. Figure 10–15 illustrates this point.

Figure 10–15. Gate-Level Register Retiming Diagram

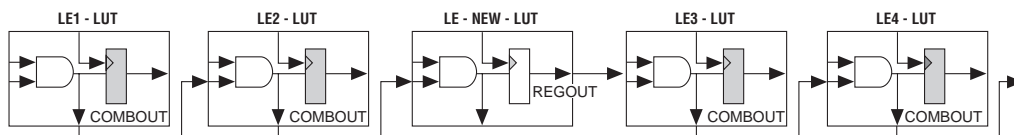
For information on how Quartus II Physical Synthesis can automatically perform gate-level retiming without altering functionality, see the *Netlist Optimizations and Physical Synthesis* chapter in Volume 2 of the *Quartus II Handbook*.

Figure 10–16 shows a design with unbalanced combinational delay. To balance the logic on either side of the combinational logic, follow the steps listed below:

Figure 10–16. Combinational Logic Before Using Chip Editor

1. Create a new LE using the Chip Editor (LE-NEW).
2. Connect the COMBOUT port of LE2 to the DATAIN port of LE-NEW.
3. Connect the REGOUT port of LE-NEW to the input of LE3.

Figure 10–17 shows the design with balanced combinational delay.

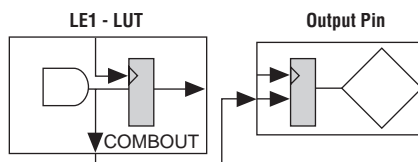
Figure 10–17. Combinational Logic After Using Chip Editor

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 10–18](#)).

Figure 10–18. Routing an Internal Signal to an Output Pin

1. Create an output pin.
2. Create the REGOUT or COMBOUT of 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.

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 10–7](#).

Table 10–7. Timing Analysis Data (Part 1 of 2)

Slack	Required t_{CO}	Actual t_{CO}	From	To	From CLK
-0.317 ns	7.000 ns	7.317 ns	outff_a~7	Out	clk
-0.204 ns	7.000 ns	7.204 ns	outff_a~6	Out	clk

Table 10–7. Timing Analysis Data (Part 2 of 2)

Slack	Required t_{CO}	Actual t_{CO}	From	To	From CLK
-0.136 ns	7.000 ns	7.136 ns	outff_a~8	Out	clk
-0.008 ns	7.000 ns	7.008 ns	outff_a~9	Out	clk

The equation for t_{CO} is defined as:

$$t_{CO} = \text{<clock to source register delay>} + \text{<micro clock to output delay>} + \text{<register to pin delay>}$$

To meet the t_{CO} requirement, either the <clock-to-source register delay> or the <register-to-pin delay> (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 -318 ps for clock clk between source register outff_a~9 and destination pin out

```

Info: - tco from clock to output pin is 7.318 ns
Info: + Longest clock path from clock clk to source register is 2.684 ns
      Info: 1: + IC(0.000 ns) + CELL(0.619 ns) = 0.619 ns; Loc. = Pin_L2; Fanout =
100; CLK Node = 'clk'
      Info: 2: + IC(1.523 ns) + CELL(0.542 ns) = 2.684 ns; Loc. = LC_X32_Y30_N2;
Fanout = 1; REG Node = 'outff_a~9'
      Info: Total cell delay = 1.161 ns ( 43.26 % )
      Info: Total interconnect delay = 1.523 ns ( 56.74 % )
Info: + Micro clock to output delay of source is 0.156 ns
Info: + Longest register to pin delay is 4.478 ns
      Info: 1: + IC(0.000 ns) + CELL(0.000 ns) = 0.000 ns; Loc. = LC_X32_Y30_N2;
Fanout = 1; REG Node = 'outff_a~9'
      Info: 2: + IC(0.400 ns) + CELL(0.366 ns) = 0.766 ns; Loc. = LC_X32_Y30_N8;
Fanout = 1; COMB Node = 'xx[0]~190'
      Info: 3: + IC(1.093 ns) + CELL(2.619 ns) = 4.478 ns; Loc. = Pin_J9; Fanout =
0; PIN Node = 'out'
      Info: Total cell delay = 2.985 ns ( 66.66 % )
      Info: Total interconnect delay = 1.493 ns ( 33.34 % )

```

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 can 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

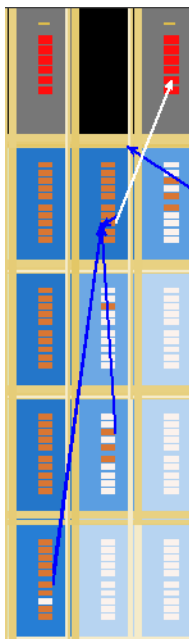
the register-to-pin delay, thereby reducing the t_{CO} . It should be noted that by moving the register, the f_{MAX} of the overall circuit may decrease. Also, to use the manual gate-level retiming process you must ensure that moving the register does not alter the functionality of the circuit. In general, this method should only be used 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.

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 10–19).

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 10–19. Failing Path in Chip Editor



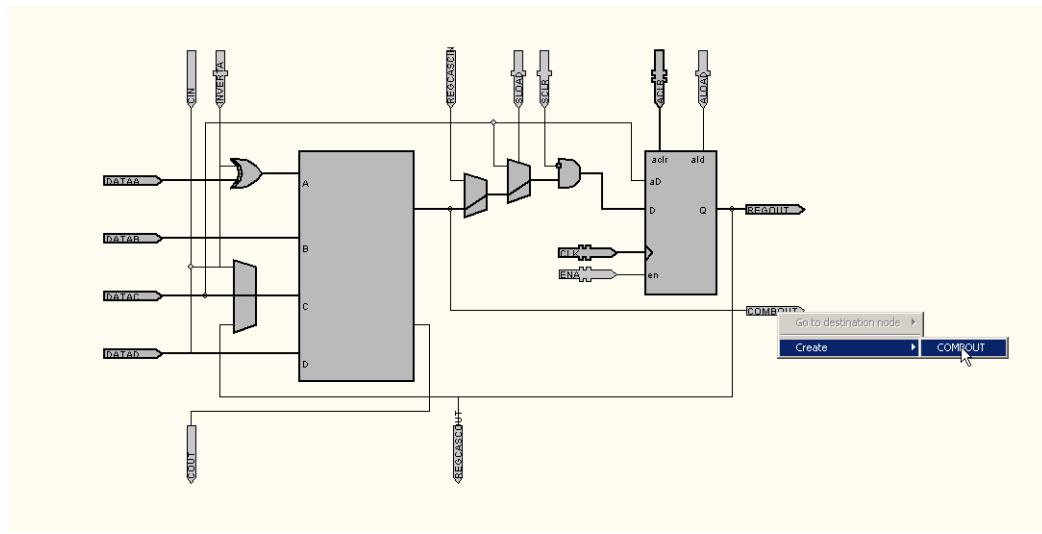
2. Open the Resource Property Editor and locate the source register.

Right click on the source register (outff_a~9) and select **Locate in Resource Property Editor** (right button pop-up menu).

3. Create the COMBOUT port for outff_a~9.

Right click the COMBOUT port and select **Create COMBOUT** (right button pop-up menu). See Figure 10-20.

Figure 10-20. Select Create COMBOUT



4. Connect COMBOUT of outff_a~9 to DATA input of xx[0]~190.

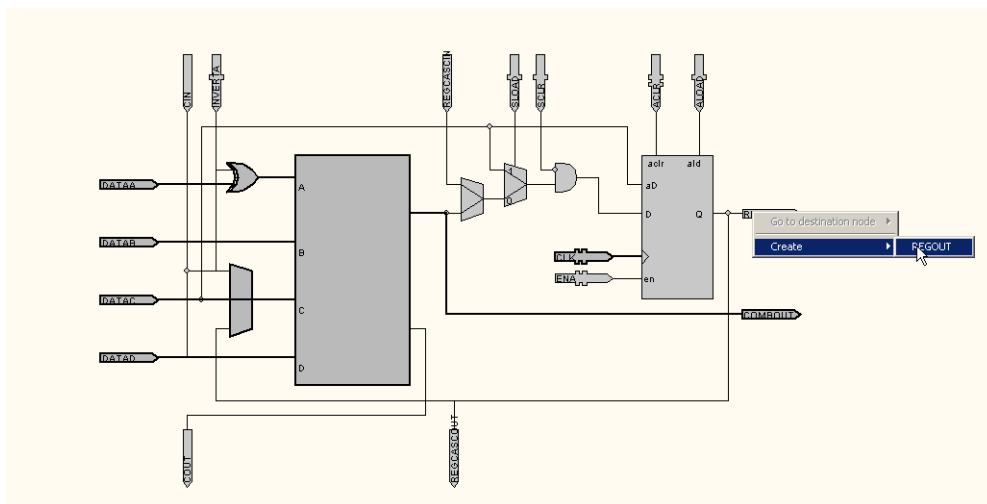
To perform this step, you must perform a **Check & Save All Netlist Changes** from the Change Manager to ensure that the newly created COMBOUT port for outff_a~9 appears in the Node Finder.

- a. Right click in the **Change Manager** and select **Check & Save All Netlist Changes**.
- b. Open the **Resource Property Editor** for xx[0]~190.
- c. Right click on the DATA port of xx[0]~190 and select **Edit Connections**. In the **Edit Connections** dialog box, find the COMBOUT port outff_a~9 with the **Node Finder**.

We 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 in `clk` (the name of the system clock in the design).
 - b. Right click on the REGOUT port and select **Create REGOUT** (see Figure 10–21).

Figure 10–21. Select Create REGOUT

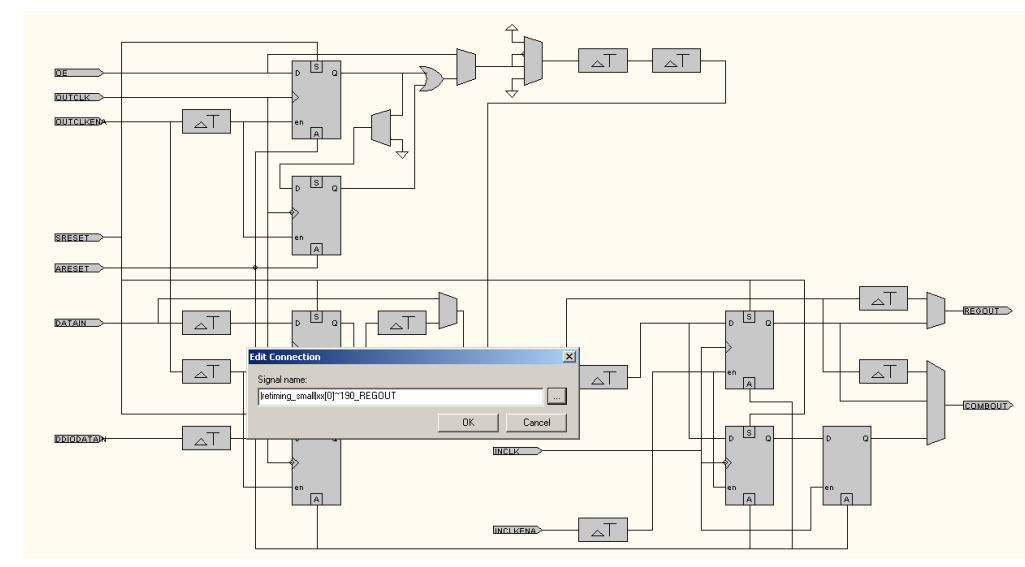


6. Remove connection between COMBOUT of xx[0]~190 and DATAIN of out.
 - a. Right click on the COMBOUT of xx[0]~190 and select **Go To Destination atom out**.
 - b. Right click on the DATAIN port of out and select **Remove Connection**.
7. Connect REGOUT to DATAIN of the output pin-out.

To perform this step you must run the **Check & Save All Netlist Changes** command in the **Change Manager** to ensure that the newly created REGOUT from step 6 for xx[0]~190 appears in the Node Finder.

- Right click in the **Change Manager** and select **Check & Save All Netlist Changes**.
- Open the **Resource Property Editor** for out.
- 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 10-22](#).

Figure 10-22. Select Edit Connections



- 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 10–8 describes the Timing Analysis report after the changes have been made.

<i>Table 10–8. Timing Analysis Report After Changes Have Been Made</i>					
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 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 element, 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.

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 JTAG interface, making it easier to test changes to memory contents.

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 update memory 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 logic analyzer, this feature provides you the visibility required to debug your design in the hardware lab.



For more information on SignalTap II, see the *Design Debugging Using the SignalTap II Embedded Logic Analyzer* chapter 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 RAMs 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 version 4.1.

Table 11–2 lists the types of memory supported by the MegaWizard Plug-In Manager and the In-System Memory Content Editor.

Table 11–1. 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 11–2 lists support for in-system updating of memory and constants for the APEX™ 20K, APEX II, Mercury™, Stratix®, and Cyclone™ device families.

Table 11–2. Supported Megafunctions							
MegaFunction	APEX 20K	APEX II	Mercury	Stratix M512 blocks	Stratix M4K blocks	Stratix MegaRAM blocks	Cyclone
LPM_CONSTANT	Read/Write	Read/Write	Read/Write	Read/Write	Read/Write	Read/Write	Read/Write
LPM_ROM	Write	Read/Write	Read/Write	Write	Read/Write	N/A	Read/Write
LPM_RAM_DQ	N/A (1)	Read/Write	Read/Write	Read/Write	Read/Write	Read/Write	Read/Write
ALTSYNCRAM (ROM)	N/A	N/A	N/A	N/A	Read/Write	Read/Write	Read/Write
ALTSYNCRAM (Single-Port RAM Mode)	N/A	N/A	N/A	Read/Write	Read/Write	Read/Write	Read/Write

Note to Table 11–2:

- (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 configurable.
3. Perform a full compilation.
4. Program your device.

Creating In-System Configurable Memory and Constants

When you enable a memory or constant to be run-time configurable, 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 configuration without changing the functionality of your design. For a list of run-time configurable megafunctions, refer to [Table 11-1](#).

To enable your memory or constant to be configurable, 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. In addition to 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 for **Instance ID**. These parameters can be changed on the last page of the wizards for megafunctions that support in-system updating.
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>"
```

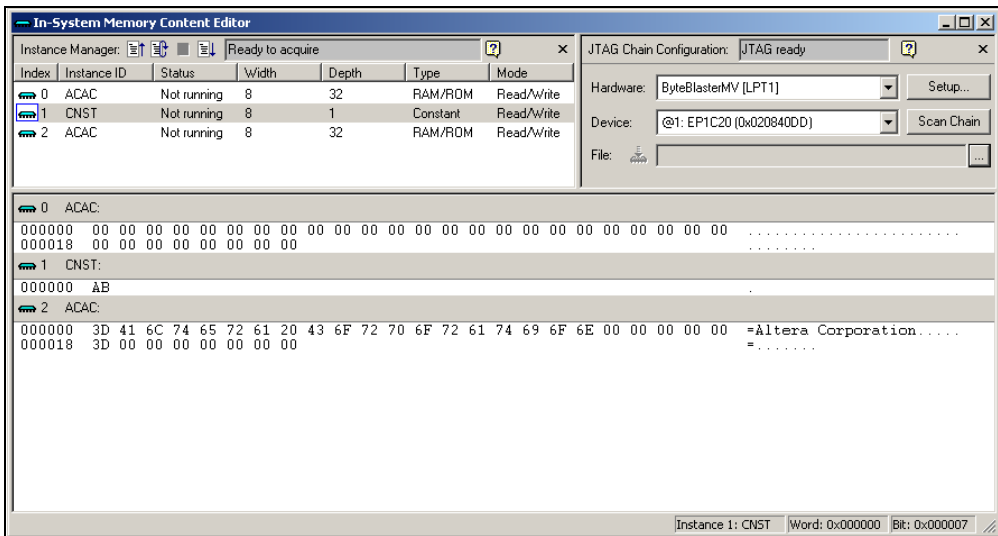
In Verilog HDL code, add the following:

```
<megafunction>_component.lpm_hint = "ENABLE_RUNTIME_MOD
= YES, INSTANCE_NAME=<instantiation name>"
```

Running the In-System Memory Content Editor

The In-System Memory Content Editor is separated into the Instance Manager, JTAG Chain Configuration and the Hex Editor (Figure 11–1).

Figure 11–1. In-System Memory Content Editor



The Instance Manager displays all available run-time configurable 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. Enter and evaluate data in the Hex Editor.

Using the In-System Memory Content Editor does not require you to 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.

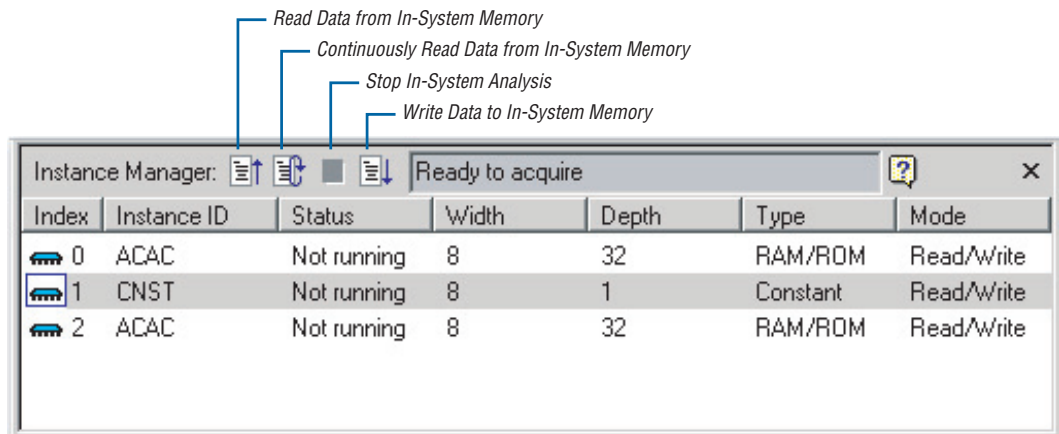
The In-System Memory Content Editor can modify the contents of memory in a single device. If you have more than one device containing in-system configurable memories or constants in a JTAG chain, you can launch multiple In-System Memory Content Editors within the Quartus II software to access the memories 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 configurable 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 11–2](#).

Figure 11–2. 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*—Stops the current read or write operation
- *Write Data to In-System Memory*—Asynchronously writes data present in the Hex Editor to the device

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 11–3).

Figure 11–3. Compilation Report In-System Memory Content Editor Setting Section

new_ver16 Compilation Report

Compilation Report

- Legal Notice
- Flow Summary
- Flow Settings
- Flow Elapsed Time
- Flow Log
- Analysis & Synthesis
 - Analysis & Synthesis Summary
 - Settings
 - Analysis & Synthesis Optimization Results
 - Analysis & Synthesis Debug Settings Summary
 - SignalTap II Logic Analyzer Settings
 - In-System Memory Content Editor Setting**
 - Hierarchy
 - Analysis & Synthesis Resource Utilization by Entity
 - Analysis & Synthesis Equations
 - Analysis & Synthesis Source Files Read
 - Analysis & Synthesis Resource Usage Summary
 - Analysis & Synthesis RAM Summary
 - Analysis & Synthesis Messages
- Filter
- Assembler
- Timing Analyzer

In-System Memory Content Editor Setting

Instance Index	Instance ID	Width	Depth	Mode	Hierarchy Location	
1	0	ACAC	16	32	Read/Write	ltoplmy_8x32ram_dq:inst1altsynkra...
2	1	CNST	8	1	Read/Write	ltoplmy_constant:instlpm_constant.L...
3	2	ACAC	16	32	Read/Write	ltoplmy_8x32ram_dq:inst3altsynkra...

Making Changes

To read the contents of in-system memory, click **Read Data from In-System Memory** or **Continuously Read Data from In-System Memory** in the Instance Manager. You can also run these commands by right-clicking in the Instance Manager or Hex Editor and choosing from the right button pop-up menu.

To edit data before writing it back to the device, place the insertion point at the desired location in the Hex Editor and begin typing. Editing always overwrites data in the hex editor. Modified data appears in blue until it is written, when it appears red.

To prepare data, type or paste changes into the Hex Editor or import a memory file. The In-System Memory Content Editor supports importing of hexadecimal (.hex) and memory initialization file (.mif) formats.

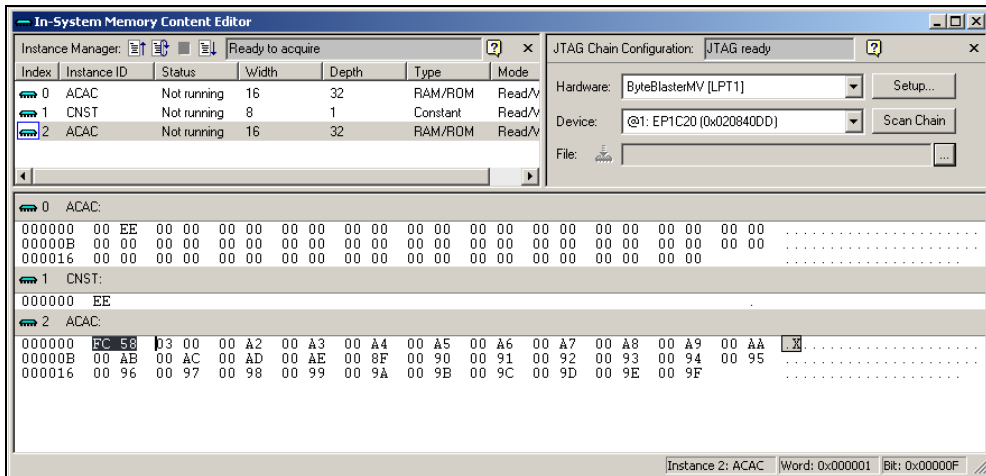
To import a file, right-click the target instance in the Instance Manager or a specific location in the Hex Editor and choose **Import Data from File** in the Instance Manager. The file data overwrites the data displayed at the chosen location in the Hex Editor.

After reading data from in-system memory, export it to a file by right mouse clicking the instance in the Instance Manager or the data in the Hex Editor and choosing **Export Data to File**. You can export data to HEX, MIF, Value Change Dump (.vcd), or RAM Initialization file (.rif) format.

Viewing Memory & Constants in the Hex Editor

For each instance of an in-system memory or constant, the Hex Editor displays data in hexadecimal numbers 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 11–4).

Figure 11–4. Editing 16-bit Memory Words Using the Hex Editor



Unprintable ASCII characters are represented by a period (.). The color of the data changes in color 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 the bit offset (Figure 11-5).

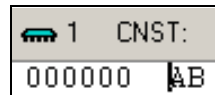
Figure 11-5. Status Bar in the In-System Memory and Content Editor



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.

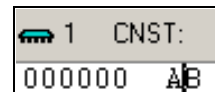
With the cursor in the position shown in Figure 11-7, the word location is 0x0000 and the bit position is 0x0007.

Figure 11-6. Hex Editor Cursor Positioned at Bit 0x0003



With the cursor in the position shown in Figure 11-6, the word location remains 0x0000 but the bit position is 0x0003.

Figure 11-7. Hex Editor Cursor Positioned at Bit 0x0007



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 SOF file 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**.

Conclusion

The In-System Updating of Memory and Constants feature and In-System Memory Content Editor provides access into a device for efficient debug in a hardware lab. You can use In-System Memory Updating of Memory and Constants with SignalTap II to maximize the visibility into an Altera FPGA. The more visibility and access to the internal logic of the device that you have, the quicker problems can be identified and resolved.

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, Cadence Incisive Conformal Support](#)

Revision History

The table below shows the revision history for [Chapter 12](#).

Chapter(s)	Date / Version	Changes Made
12	June 2004 v2.0	<ul style="list-style-type: none"> • Updates to tables, figures. • New functionality for Quartus 4.1. • This chapter was formerly chapter 11 in the previous section.
	Feb. 2004 v1.0	Initial release.

Introduction

The Altera® Quartus® II software version 4.1 easily interfaces with EDA tools such as the Cadence Incisive Conformal software and Synplicity Synplify software. In addition, the Quartus II software has built-in support for verifying the logical equivalence between the synthesized (.vqm) netlist from Synplicity Synplify and the post-fit Verilog (.vo) files using the Incisive Conformal software.

This chapter discusses the following topics:

- Formal verification
- Setting up the Quartus II software to generate the VQM file and Incisive Conformal script
- Comparing designs using Incisive Conformal software
- Known issues and 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.



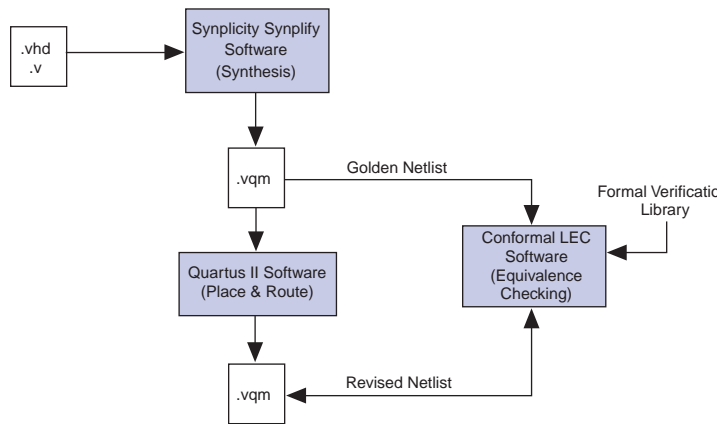
The formal verification flow can be used for designs targeting the Cyclone™, Stratix™ GX, and Stratix device families.

Equivalence Checking

Equivalence checking is used to compare the functional equivalence of the original design with the revised design by using mathematical techniques rather than by performing simulation using test vectors, greatly decreasing the time to verify the design.

Altera supports formal verification of the post-synthesis netlist from Synplify and Synplify Pro and the post-place-and-route netlist from Quartus II software, as shown in [Figure 12-1](#).

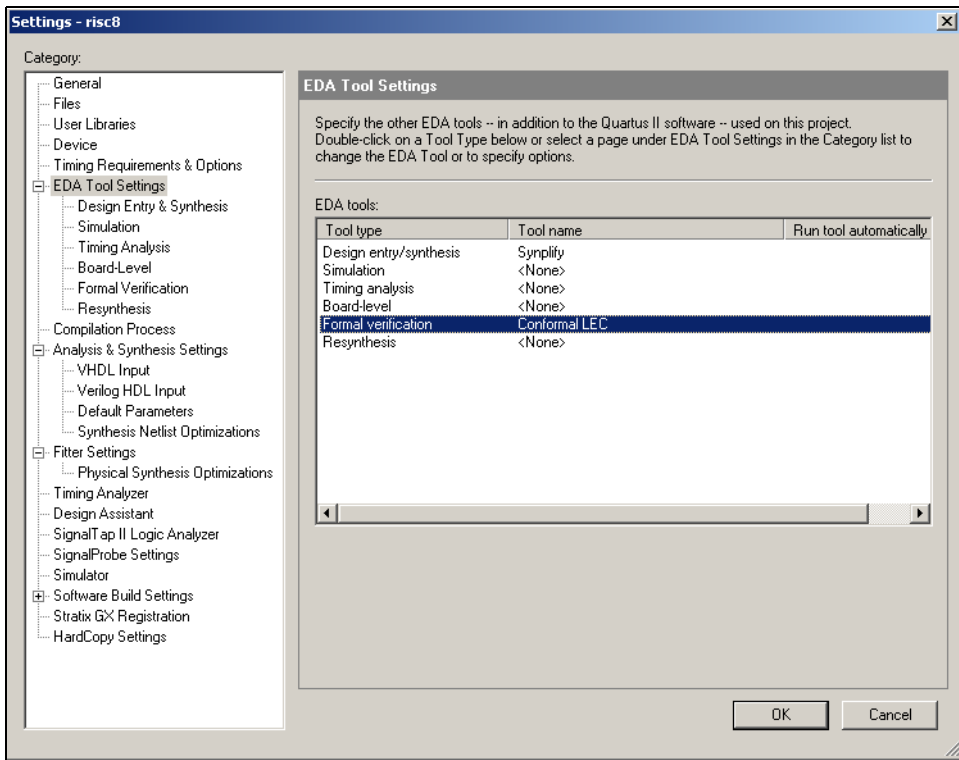
Figure 12–1. Formal Verification Flow Using Synplify & Incisive Conformal Software



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 file and Incisive Conformal script for use in formal verification:

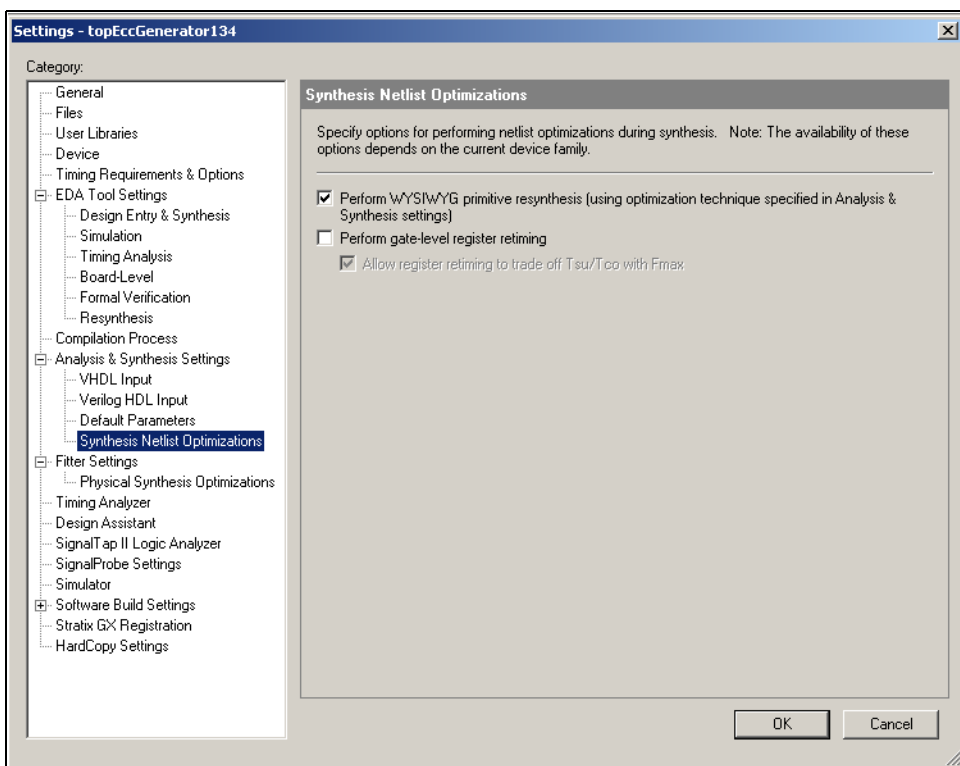
1. If you have not yet done so, create a new Quartus II project or open an existing project.
2. Choose **EDA Tools Settings** (Assignments menu).
3. On the **EDA Tool Settings** page of the **Settings** dialog box, under **EDA tools**, for **Design entry/synthesis** specify **Synplify** or **Synplify Pro**. Specify **Conformal LEC** for **Formal verification** (Figure 12–2).

Figure 12–2. EDA Tools Selection *Note (1)*

Note to Figure 12–2:

- (1) The Quartus II software allows up to six EDA tools to be selected in the EDA tools list.

4. Choose **Analysis and Synthesis** in the Category list of the **Settings** dialog box.
5. Under **Analysis and Synthesis**, select **Synthesis Netlist Optimizations**. On the **Synthesis Netlist Optimizations** page, ensure that **Perform gate-level register retiming** is turned off (Figure 12–3).

Figure 12–3. 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 (Figure 12–4).

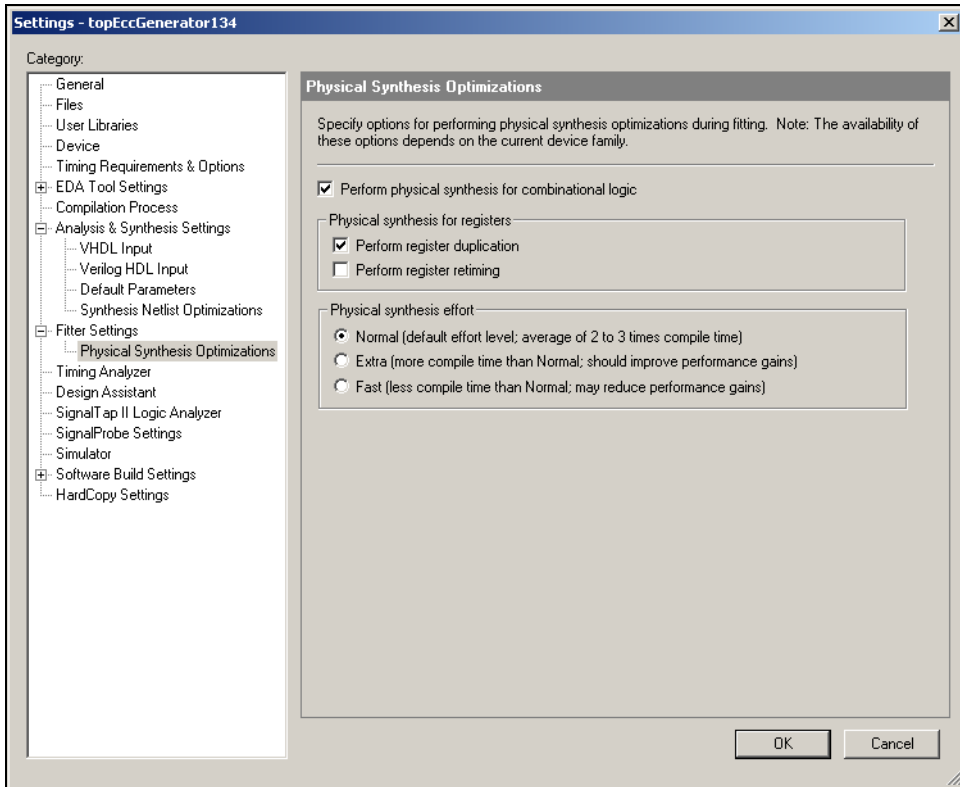


Retiming a design usually results in moving and merging registers along the critical path and is not very well supported by equivalence checking tools. Because equivalence checkers compare the cones of logic terminating at registers, it is necessary that registers not be moved during Quartus II optimization.

If the options described in this section are not selected, the Incisive Conformal script may be presented with a different set of compare points, and the resulting netlist would be difficult to compare against the reference netlist file.

The Quartus II software version 4.1 supports register duplication to improve timing results. The formal verification tool also supports register duplication and can be used during the formal verification flow (Figure 12–4).

Figure 12–4. 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 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 project includes any of the following design entities, the synthesized VQM netlist file from the Synplify software contains black boxes and their boundary interface must be preserved:

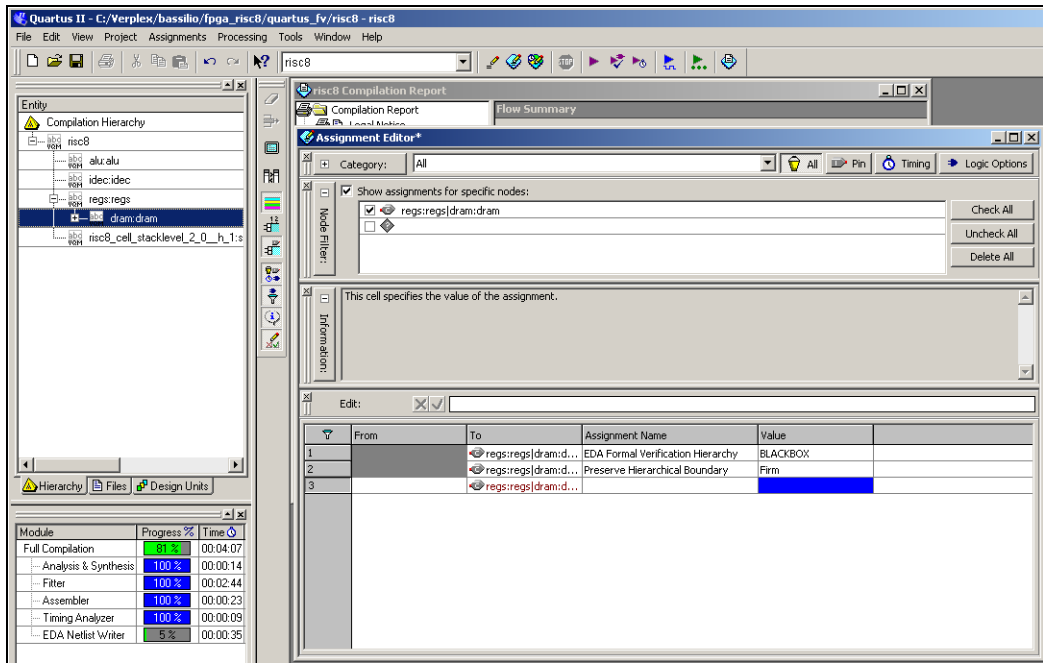
- Altera library of parameterized modules (LPMs) functions. The black box property is applied to only those LPM modules for which an equivalent Incisive Conformal model does not exist.
- Encrypted intellectual property (IP) cores.
- Entities that are defined in the design format other than Verilog HDL or VHDL.

The Quartus II software version 4.1 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.

Users can also set the black box property on the entities that need not be compared by the formal verification tool. To do so make the following assignments for the entities in question:

- An **EDA Formal Verification Hierarchy** assignment with the value **BLACKBOX**
- A **Preserve Hierarchical Boundary** assignment with the value **Firm** (Figure 12-5).

Figure 12–5. Setting the Black Box Property on a Module



The Quartus II software compiler generates:

- A VO file *<design_name>.vo*
- A Script file *<design_name>.ctc* used with Incisive Conformal software, referencing *<design_name>.clg* and *<design_name>.clr* to read the library files and black box descriptions
- A **blackboxes** directory, containing all the user-defined black box entities in the design at *<project directory>/fv/conformal/blackboxes*.

The script file contains the setup constraints to be used along with the formal verification tool. Following is the sample setup constraints generated by the Quartus II software:

```
add renaming rule r1 "_aI$" ""-revised
add renaming rule r2 "\\/" "_a" -golden
add renaming rule r3 "\\/" "_a" -revised
```

```
add ignored inputs data_b[3] data_b[2] data_b[1]
    address_b[3] address_b[2] address_b[1]
-module altsyncram_width_a8_widthad_a7
-revised
```

```
set mapping method -unreach
```

```
set mapping method -phase
```

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 as it is defined in the **altdpram.v** file of the formal verification library.

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 12–2](#), the netlist synthesized by the Quartus II software contains black boxes if your project includes any of the following:

- LPM functions
- Encrypted IP functions
- Entities not implemented in Verilog HDL or VHDL

Every LPM function is treated as a black box by the Synplify software. If a corresponding Incisive Conformal verification model exists, however, the LPM function is replaced by logic cells in the VQM 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 the user makes 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 12–3).

If the above two assignments 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 from a system command prompt type the following:

```
lec -dofile /<path to project directory>/fv/conformal/<design_name>.ctc
      -nogui ↵
```

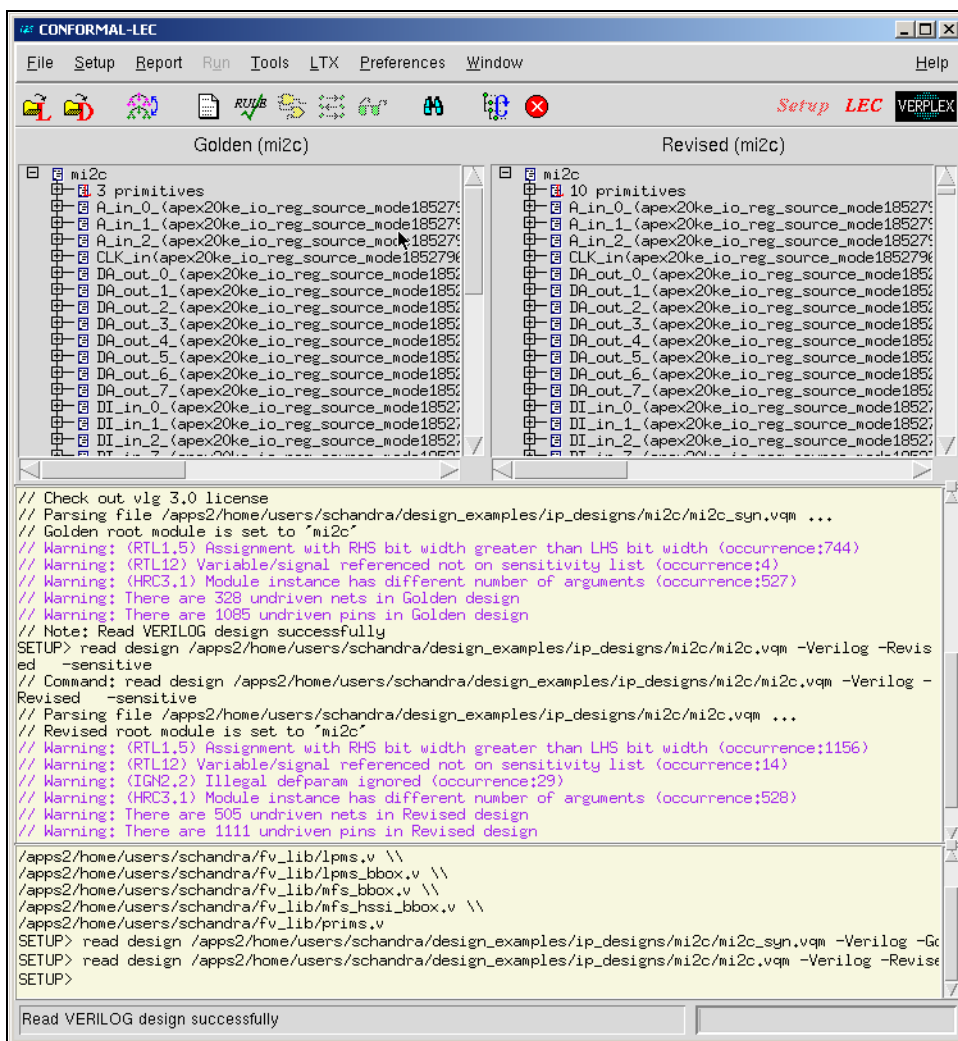
Running the Incisive Conformal Software from the GUI

To run the Incisive Conformal software using the GUI, do the following:

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 12–6. 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 12–6. 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 may be encountered when using the formal verification flow described in this chapter:

- Unused logic optimized within a black box by the Quartus II software can result in an interface different from the interface in the synthesized VQM netlist.
- In designs with combinational feedback loops, the Incisive Conformal software may incorrectly insert extra, unmapped cut points in the revised netlist.

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.

+transport_int_delays 2-8
+transport_path_delays 2-8

A

Acquisition Clock
 Assigning 9-4
Adaptive Logic Module 10-10
Add Signals
 Command-Line Mode 3-15
 GUI Mode 3-16
 to View 3-23
 to View 3-15
Advanced Timing Analysis
 Reports Using Tcl Scripts 4-34
ALM Properties 10-14
Altera Megafunction 3-8
Analyzer
 Triggering 9-6
Applications
 Common 10-24
Assigning Data Signals 9-5
Assignments
 Multicycle 4-16
 Multicycle Hold 4-17
 Multicycle Source 4-18
 Source Multicycle Hold 4-19
Asynchronous Memory 4-14

B

Bird's Eye View 10-5
Buffer Acquisition 9-23

C

Captured Data
 Converting to Other File Formats 9-22
 Saving 9-22
cds.lib 3-6
 Command-Line Mode 3-7

 GUI Mode 3-7
Change Manager 10-23
Chip Editor 10-3
 Floorplan 10-4
 Locating a Node 9-31
 Using in Design Flow 10-2
Clock
 Derived Clocks 4-13
 Frequency
 Maximum 4-3
 Hold Time 4-2
 Inverted Clock 4-10
 Not a Clock 4-11
 Output Clock Frequency
 Adjusting 10-22
 Requirements
 Specifying Individual 4-7
 Settings 4-8
 Setup Time 4-1
 Skew 4-5, 4-13
 Reduce 4-31
 to-Output Delay 4-3
Command Prompt
 2-11, 3-30
Compilation
 Command-Line Mode 3-11
 Faster 9-20
 GUI Mode 3-12
Compile
 Project Files & Libraries 3-21
 Source Code & Testbenches 3-11
Cut Off
 Clear and Preset
 Signal Paths 4-28
 Feedback
 I/O Pins 4-28
 Read During Write Signal Paths 4-29
Cut Paths Between Unrelated Clock
 Domains 4-30
Cut Timing Path 4-30

D

- Data
 - Capturing to Specific RAM Type 9–24
- Data Delay
 - Increase 4–32
- Data Samples
 - View 9–12
- Design
 - Cycle
 - Estimating Power 6–3
 - Elaborate 3–21
 - Flaw
 - Correcting 10–27
 - Simulating with Memory 3–10
- Device & Megafunction Support 11–2
- Duty Cycle
 - Adjusting 10–22
- Dynamically Link 3–24
- Dynamically Load 3–25

E

- EDA Simulation Tools
 - Estimating Power 7–4
- Elaborate Design 3–13
- Elaboration
 - Command-Line Mode 3–13
 - GUI Mode 3–14
- Elements
 - Cyclone I/O 10–17
 - Editable Properties of I/O 10–19
 - FPGA I/O 10–15
 - Stratix, Stratix GX, & Stratix II I/O 10–15
 - Supported Changes for an I/O 10–18
- Embedded Logic Analyzer
 - Creating with MegaWizard Plug-In Manager 9–8
- Embedding Multiple Analyzers in One FPGA 9–20
- Environment
 - Setting Up 3–5, 3–20
 - Setting Variables 3–5
- Equipment Setup 9–26
- Equivalence Checking 12–1

F

- False Paths 4–28
- File Conversion
 - HEX 1–10, 2–4
- FPGA Memory
 - Preserving 9–13
- Functional RTL Simulation 1–3, 1–4, 2–2
 - Altera Memory Blocks 2–3
 - Command-Line Mode 3–18
 - GUI Mode 3–18
 - Libraries 1–4
- Functional RTL Simulation 3–2, 3–5

H

- Hex Editor
 - Viewing Memory & Constants 11–7
- Hold Time Violations
 - Fixing 4–31

I

- I/O Elements
 - MAX II 10–17
- I/O Standards
 - Assigning 8–4, 8–10
- Incisive Conformal 12–8
 - Black Boxes in Flow 12–8
 - Running 12–9
 - Running from Command Prompt 12–9
 - Running from GUI 12–9
 - Script & VO File 12–2
- Instance Manager 11–5
- In-System
 - Configurable Memory and Constants 11–3
 - Memory Content Editor 11–4, 11–8
 - Updating 11–3

L

- LE/ALM
 - Supported Changes 10–11
- Libraries
 - Create 3–6, 3–20
 - LPM Function 3–8
- Libraries
 - Quartus II Timing Simulation 3–20

- Library Setup 3–6
- Licensing 1–20
- Local PC
 - Software Setup 9–27
- Logic Element 10–9
 - Properties 10–12
- LPM
 - Functional RTL Simulation Models 1–4
- LUT
 - Equation 10–12
 - Mask 10–13
- LUT Mask 10–14

M

- Maximum Delay
 - Input 4–9
 - Output 4–9
- Meeting I/O Timing 10–27
- MegaWizard-Generated File
 - Modifying 1–10, 2–4
- MIF to RIF 1–10, 2–4
- Minimum Timing Analysis 4–33
 - Performing 4–33
 - Reporting 4–34
 - Settings 4–33
- Mnemonics
 - Creating for Bit Patterns 9–23
- Mode
 - Extended LUT Mode 10–15
 - External Feedback 10–22
 - of Operation 10–12
 - Register Cascade Mode 10–14
 - Shared Arithmetic Mode 10–15
 - Synchronous Mode 10–14
- ModelSim-Altera Software 1–3
 - Quartus II Software Output Files 1–11
- Modes
 - Operation 3–3
- Modifying the PLL Using the Chip Editor 10–21
- Multicycle Assignments
 - Typical Applications 4–19
- Multicycle Hold Assignments 4–31
- Multicycle Paths
 - Multi-Frequency Domains 4–24

- Offsets 4–23
- Simple 4–19
- Multiple Clock Domains 4–15

N

- NativeLink
 - Using with ModelSim 1–19
- NC Simulation Flow 3–4
- NC-Sim
 - Generated Simulation Output Files 3–29
- Netlist
 - Generating for Other EDA Tools 10–33
- Nodes
 - Select Nodes Reserved for Incremental Routing 9–21
 - Set Number Allocated 9–20
- nopli.v
 - Compiling 1–11, 2–4

O

- Output Files
 - Quartus II Simulation 3–18

P

- Phase Shift
 - Adjusting 10–22
 - of PLL
 - Adjusting to Meet I/O Timing 10–27
- Pin-to-Pin Delay 4–3
- Pipelining
 - Adding Registers 8–4, 8–11
- PLI Routines
 - Incorporating 3–23
 - VCS
 - Software 2–10
- PLL Mode
 - External Feedback 10–23
 - Normal 10–22
- Post-Synthesis Simulation 2–4
 - Generating Netlist 2–5
- Power
 - Calculator
 - Excel-Based 6–1
 - Estimation

- Quartus II Software 7-2
- Simulation-Based Settings 7-7
- Input File
 - Generate 7-8
 - Report File 6-6
- Preserving Timing 9-32
- PrimeTime
 - Environment
 - Generated Files 5-2
 - Format
 - Specified Constraint Samples 5-4
 - Quartus II Settings to Generate Files 5-1
 - Running 5-6
 - Sample Timing Report 5-5
 - Timing Reports 5-4
- Programming File 10-33
- Properties
 - Cyclone 10-20
 - Max II 10-21
 - PLL 10-21
 - Stratix and Stratix GX 10-19
 - Stratix II 10-19

Q

- Quartus II
 - Megafunction
 - Simulation Models 1-4

R

- Register Retiming
 - Gate-Level 10-24
- Remote PC
 - Software Setup 9-26
- Resource Property Editor 10-9
- Routing Internal Signal to Output Pin 10-26

S

- Sample Depth
 - Specifying 9-6
- Scripting Support 2-10, 3-29, 7-7, 8-9
- SDF Command File 3-22
- Signal Preservation 9-28
- SignalProbe
 - Adding Sources 8-3, 8-10

- Compilation
 - Performing 8-5
- Fitting Results Modification 8-12
- Pins
 - Reserving 8-2, 8-10
- Results Compilation 8-8
- Routing
 - Enable or Disable All 8-11
- Routing Failures 8-7
- Run Automatically 8-11
- Run Manually 8-11
- Running with Smart Compilation 8-7, 8-12
- Using 8-1
- SignalTap II
 - Analysis
 - Programming Device 9-12
 - Logic Analyzer
 - Compiling Design 9-8
 - Creating HDL Representation 9-8
 - Debug Multiple Designs 9-29
 - Including in Design 9-2
 - Instantiating in HDL 9-11
 - Timing Preservation 9-29
 - Logic Analyzers
 - SOPC Builder Systems 9-35
- SignalTap II
 - Local PC Setup 9-28
 - Megafunction Ports 9-11
 - Remote Debugging 9-25
 - Used FPGA Resources 9-24
 - Using in Lab Environment 9-25
- Simulate
 - Design 3-23
 - Design 3-17
- Simulation
 - Flow 3-1
- Libraries
 - Gate Level 1-12
- Slack 4-4
 - Hold Time 4-4
- Spread Spectrum
 - Adjusting 10-23
- Standard Delay Output File
 - Compiling 3-22
- Statically Link 3-28
- STP File
 - Assigning Signals 9-5

- Creating 9-3
 - Using to Create Embedded Logic Analyzer 9-3
- ## T
- Tappable Signals 9-29
 - Tcl
 - Commands 2-11
 - commands 3-29
 - Executing Script-Based Commands 4-6
 - tCO Requirement 4-11
 - Testbench
 - Compile into Work Library 1-18
 - tH Requirement 4-12
 - Time Bars and Next Transition 9-22
 - Timing
 - Analysis
 - Advanced 4-13
 - Asynchronous Domains 4-32
 - Basics 4-1
 - Third-Party Software 4-34
 - Analysis Reporting 4-12
 - Analyzer 4-6
 - Running 10-33
 - Assignments
 - Setting Global 4-7
 - Setting Other Individual 4-8
 - Simulation
 - Gate-Level 1-4, 1-11, 2-6
 - Generating Gate-Level Netlist 2-6
 - Simulation Netlist
 - Gate-Level for VCS 2-11
 - Simulation
 - Gate-Level 3-3, 3-18
 - Wizard 4-12
 - tPD Requirement 4-12
 - Transport Delays 2-8
 - Trigger
 - Creating Complex 9-14
 - In 9-17
 - Levels
 - Number of 9-7
 - Out 9-17
 - Using as Trigger In of Another Analyzer 9-18
 - Position Specifying 9-7
 - Type
 - Basic or Advanced 9-6
 - Using External 9-17
 - tSU Requirement 4-12
- ## V
- Variable
 - LM_LICENSE_FILE 1-20
 - VCS
 - Compile Switches
 - Common 2-8
 - Debugging VCS
 - Command-Line Interface 2-9
 - Netlist
 - Generating Post-Synthesis Simulation 2-11
 - Using in Quartus II Design Flow 2-1
 - Verilog
 - Code
 - Preparing & Linking C Programs 2-10
 - Functional RTL Simulation with Altera Memory Blocks 1-10
 - Simulating Designs 1-7
 - Simulation Designs 1-17
 - Verilog HDL
 - 3-11
 - Code 3-15
 - veriusers.c
 - Modified 3-26
 - Original 3-25
 - VHDL
 - 3-11
 - Simulating Designs 1-5, 1-15
 - View
 - First Level View 10-6
 - Second Level View 10-7
 - Third Level View 10-8
 - VirSim
 - Using 2-9

