

**Design Compiler®
Optimization
Reference Manual**

Version H-2013.03, March 2013

SYNOPSYS®

Copyright Notice and Proprietary Information

Copyright © 2013 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at <http://www.synopsys.com/Company/Pages/Trademarks.aspx>.

All other product or company names may be trademarks of their respective owners.

Synopsys, Inc.
700 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Copyright Statement for the Command-Line Editing Feature

Copyright © 1992, 1993 The Regents of the University of California. All rights reserved. This code is derived from software contributed to Berkeley by Christos Zoulas of Cornell University.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement: This product includes software developed by the University of California, Berkeley and its contributors.
4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright Statement for the Line-Editing Library

Copyright © 1992 Simmule Turner and Rich Salz. All rights reserved.

This software is not subject to any license of the American Telephone and Telegraph Company or of the Regents of the University of California.

Permission is granted to anyone to use this software for any purpose on any computer system, and to alter it and redistribute it freely, subject to the following restrictions:

1. The authors are not responsible for the consequences of use of this software, no matter how awful, even if they arise from flaws in it.
2. The origin of this software must not be misrepresented, either by explicit claim or by omission. Since few users ever read sources, credits must appear in the documentation.
3. Altered versions must be plainly marked as such, and must not be misrepresented as being the original software. Since few users ever read sources, credits must appear in the documentation.
4. This notice may not be removed or altered.

Contents

About This Manual	xvi
Customer Support.	xix
1. Basic Concepts for Optimizing Designs	
Using DC Ultra	1-3
Exploring the Design Space	1-3
Optimization Phases.	1-4
Combinational Optimization.	1-5
Technology-Independent Optimization	1-6
Mapping	1-6
Technology-Specific Optimization.	1-7
Initial Sequential Optimization	1-7
Final Sequential Optimization	1-8
Local Optimizations	1-9
Optimization Flow	1-9
Automatic Ungrouping	1-11
High-Level Optimization and Datapath Optimization	1-11
Multiplexer Mapping and Optimization.	1-11
Finite State Machine Optimization	1-11
Sequential Mapping	1-12
Structuring and Mapping	1-12
Auto-Uniquification.	1-12
Implementing Synthetic Parts	1-12
Timing-Driven Combinational Optimization	1-13

Register Retiming	1-13
Delay Optimization.	1-13
Design Rule Fixing.	1-14
Area Optimization	1-14
 2. Compiling a Design	
Before You Start	2-3
The compile_ultra Command	2-3
The compile Command.	2-5
Controlling Mapping Optimization	2-5
Customizing the compile Command	2-5
Performing High-Effort Synthesis	2-6
Performing an Incremental Compile	2-6
Performing Test-Ready Compile	2-7
Controlling Design Rule Fixing	2-7
Performing a Top-Level Compile	2-8
Using the -top Option With Other Compile Options.	2-8
Limiting Optimization to Paths Within a Specific Range	2-9
Fixing Timing Violations For All Paths	2-9
Choosing a Compile Strategy	2-9
Mixing Compilation Strategies	2-9
Using the Top-Down Hierarchical Compile Strategy	2-10
Running a Top-Down Hierarchical Compile Strategy	2-11
Using the Compile-Characterize-Write Script-Recompile Strategy	2-13
Redirecting the Output of Commands	2-16
Checking the Compile Log	2-17
 3. Optimization Techniques	
Optimizing for Delay	3-3
Optimizing for Area	3-4
Creating Path Groups	3-4
Optimizing Near-Critical Paths	3-5

Fixing Heavily Loaded Nets	3-6
Performing High-Effort Compile	3-7
Performing a High-Effort Incremental Compile	3-8
Disabling Total Negative Slack Optimization	3-8
Optimizing Across Hierarchical Boundaries	3-8
Isolating Input and Output Ports	3-9
Propagating Constants	3-12
Enabling Critical Path Resynthesis	3-12
Logic Duplication and Mapping to Wide-Fanin Gates	3-13
Removing Hierarchy	3-13
Optimizing for Multiple Clocks Per Register	3-14
Preserving the Clock Network After Clock Tree Synthesis	3-16
Optimizing Once for Best- and Worst-Case Conditions	3-18
Constraint-Related Commands	3-18
Reporting Commands	3-19
Optimizing With Multiple Libraries	3-20
Fixing Nets Connected to Multiple Ports	3-21
Building a Balanced Buffer Tree	3-22
Defining a Signal for Unattached Master Clocks	3-24
4. Optimizing Across Hierarchical Boundaries	
Boundary Optimization	4-2
Disabling Boundary Optimization Throughout the Design	4-3
Disabling Boundary Optimization for a Specific Design	4-3
Controlling Constant Propagation	4-5
Enabling Constant Propagation When Boundary Optimization is Disabled	4-5
Disabling Constant Propagation Through Specific Hierarchical Pins	4-5
Controlling Phase Inversion	4-5
Port Punching in Design Compiler Graphical	4-6
Port Punching and Phase Inversion With Automatic High-Fanout Synthesis	4-7

Other Optimizations That Affect Hierarchical Boundaries.	4-7
Automatic Ungrouping	4-7
Automatic Ungrouping of Hierarchies	4-8
Automatic Ungrouping of Designs With Timing Exceptions.	4-9
Exceptions to Automatic Ungrouping.	4-10
5. High-Level Optimization and Datapath Optimization	
Design Compiler Arithmetic Optimization	5-3
Synthetic Operators	5-4
Checking DesignWare Licenses	5-5
High-Level Optimizations	5-6
Tree Delay Minimization and Arithmetic Simplifications.	5-6
Resource Sharing	5-6
Common Subexpression Elimination	5-7
Sharing Mutually Exclusive Operations	5-8
Enhanced Resource Sharing	5-9
Datapath Optimization With DC Ultra	5-9
Enabling DC Ultra Datapath Optimization	5-10
Datapath Extraction	5-10
Datapath Implementation.	5-12
Advanced Datapath Transformations with DC Ultra.	5-12
Reporting Resources and Datapath Blocks	5-13
Analyzing Datapath Extraction	5-15
6. Multiplexer Mapping and Optimization	
Inferring SELECT_OPs.	6-3
Inferring MUX_OPs.	6-4
Library Cell Requirements for Multiplexer Optimization	6-7
Mapping Multiplexers on Asynchronous Signal Lines.	6-7
Mapping to One-Hot Multiplexers	6-8
Inferring One-Hot Multiplexers	6-8
Library Requirements for One-Hot Multiplexers.	6-9
Optimization of One-Hot Multiplexers	6-11

7. Multibit Component Optimization

Mapping to Multibit Registers	7-2
Creating Multibit Components	7-4
Removing Multibit Components	7-4
Reporting Multibit Components	7-5
Using Multibit Register Mapping With an Incremental Compile	7-6
Library Cell Requirements for Multibit Optimization	7-6

8. Optimizing Finite State Machines

Basic Description of Finite State Machines	8-2
General Behavior of a Finite State Machine	8-2
Finite State Machine Architecture	8-3
State Vector, State Encodings, and Encoding Styles.	8-3
State Vector	8-3
State Encodings	8-4
State Encoding Styles.	8-5
Completely and Incompletely Specified Finite State Machines.	8-6
Synthesizing Finite State Machines	8-7
Finite State Machine Design File Requirements	8-8
DC Ultra Automatic Methodology.	8-9
How Design Compiler Processes a Finite State Machine in the DC Ultra Automatic Flow.	8-9
The Finite State Machine DC Ultra Automatic Flow	8-11
Verifying a Finite State Machine	8-13
Creating Finite State Machine Reports.	8-13

9. Sequential Mapping

Register Inference.	9-3
Directing Register Mapping	9-6
Specifying the Default Flip-Flop or Latch	9-6
Reporting Register Types	9-7
Reporting the Register Type Specifications for the Design	9-7
Reporting the Register Type Specifications for Cells.	9-7

Unmapped Registers in a Compiled Design	9-8
Automatically Removing Unnecessary Registers	9-9
Removing Unconnected Registers	9-9
Eliminating Constant Registers	9-9
Merging Equal and Opposite Registers	9-11
Inverting the Output Phase of Sequential Elements	9-12
Mapping to Falling-Edge Flip-Flops	9-13
Resizing Black Box Registers	9-15
Preventing the Exchange of the Clock and Clock Enable Pin Connections	9-16
Mapping to Registers With Synchronous Reset or Preset Pins	9-16
Performing Test-Ready Compile	9-20
Overview of Test-Ready Compile	9-21
Scan Replacement	9-22
Selecting a Scan Style	9-24
Mapping to Libraries Containing Only Scan Registers	9-25
Mapping to the Dedicated Scan-Out Pin	9-25
Automatic Identification of Shift Registers	9-26
Using Register Replication to Solve Timing QoR, Congestion, and Fanout Problems	9-31
10. Adaptive Retiming	
Comparing optimize_registers With compile_ultra -retime	10-2
Adaptive Retiming Examples	10-2
Performing Adaptive Retiming	10-6
Controlling Adaptive Retiming	10-6
Reporting the dont_retime Attribute	10-7
Removing the dont_retime Attribute	10-7
Verifying Retimed Designs	10-7
11. Gate-Level Optimization	
Compile Cost Function	11-2
Design Rules Cost Function	11-2

Calculating Transition Time Cost	11-3
Calculating Fanout Cost	11-3
Calculating Capacitance Cost	11-3
Optimization Constraints Cost Function	11-4
Calculating Maximum Delay Cost	11-4
Calculating Minimum Delay Cost	11-6
Calculating Maximum Power Cost	11-7
Calculating Maximum Area Cost	11-7
Changing the Cost Function	11-8
Reordering the Default Priority of Constraints	11-8
Disabling the Cost Function	11-10
Prioritizing Area Over Total Negative Slack	11-10
Compile Log	11-10
Delay Optimization	11-12
Design Rule Fixing	11-14
Area Recovery	11-15
12. Incremental ASCII Flow With a Third-Party DFT Flow Example	
Performing Initial Compilation Using the compile.tcl Script	12-3
Performing Incremental Compilation Using the compile_incr.tcl Script	12-4
Using the Netlist SCANDEF Flow	12-5
Identifying the Scan Chain Structure	12-6
Performing Scan Chain Reordering in Design Compiler Graphical	12-9
Writing Out the New Scan Chain Structure	12-9
13. Verifying Functional Equivalence	
Using Formality	13-2
Adjusting Optimization for Successful Verification	13-3
Using the set_verification_priority Command	13-3
Using Single-Pass Verification	13-4
Using Third-Party Formal Verification Tools	13-5
14. Latch-Based Design Code Examples	
SR Latch	14-2

VHDL and Verilog Code Examples for SR Latch	14-2
Inference Report for an SR Latch	14-3
Synthesized Design for an SR Latch	14-4
D Latch	14-4
VHDL Code for a D Latch	14-5
Inference Report for a D Latch	14-5
Synthesized Design for a D Latch	14-6
D Latch With Asynchronous Reset	14-6
VHDL and Verilog Code for a D Latch With Asynchronous Reset	14-6
Inference Report for a D Latch With Asynchronous Reset	14-8
Synthesized Design for a D Latch With Asynchronous Reset	14-8
D Latch With Asynchronous Set and Reset	14-9
VHDL and Verilog Code for a D Latch With Asynchronous Set and Reset	14-9
Inference Report for a D Latch With Asynchronous Set and Reset	14-11
Synthesized Design for a D Latch With Asynchronous Set and Reset	14-12
D Latch With Enable (avoiding clock gating).	14-13
VHDL and Verilog Code for a D Latch With Enable	14-13
Inference Report for a D Latch With Enable	14-14
Synthesized Design for a D Latch With Enable	14-15
Inferring Gated Clocks	14-15
Case 1	14-15
Case 2	14-16
Synthesized Design With Enable and Gated Clock	14-17
D Latch With Enable and Asynchronous Reset	14-18
VHDL and Verilog Code for a D Latch With Enable and Asynchronous Reset	14-18
Synthesized Design for a D Latch With Enable and Asynchronous Reset	14-20
D Latch With Enable and Asynchronous Set	14-20
VHDL and Verilog Code for a D Latch With Enable and Asynchronous Set	14-20
Synthesized Design for D Latch With Enable and Asynchronous Set	14-23
D Latch With Enable and Asynchronous Set and Reset	14-23
VHDL and Verilog Code for D Latch With Enable and Asynchronous Set and Reset	14-23

Synthesized Design for D Latch With Enable and Asynchronous Set and Reset	14-26
--	-------

Index

Preface

This preface includes the following sections:

- [About This Manual](#)
- [Customer Support](#)

About This Manual

The *Design Compiler Optimization Reference Manual* describes concepts and commands used for optimizing designs and performing timing analysis using Design Compiler.

Audience

This manual is intended for logic designers and engineers who use the Synopsys synthesis tools to design ASICs and ICs. Knowledge of high level techniques, a hardware description language, such as VHDL or Verilog is required. A working knowledge of UNIX is assumed.

Related Publications

For additional information about Design Compiler, see the documentation on SolvNet at the following address:

<https://solvnet.synopsys.com/DocsOnWeb>

You might also want to see the documentation for the following related Synopsys products:

- Design Vision
- DesignWare components
- DFT Compiler and DFTMAX
- DC Explorer
- PrimeTime
- Power Compiler
- HDL Compiler
- IC Compiler

Release Notes

Information about new features, changes, enhancements, known limitations, and resolved Synopsys Technical Action Requests (STARs) is available in the *Design Compiler Release Notes* in SolvNet.

To see the *Design Compiler Release Notes*,

1. Go to the Download Center on SolvNet located at the following address:
<https://solvnet.synopsys.com/DownloadCenter>
2. Select Design Compiler, and then select a release in the list that appears.

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
Courier	Indicates syntax, such as <code>write_file</code> .
<i>Courier italic</i>	Indicates a user-defined value in syntax, such as <code>write_file design_list</code> .
Courier bold	Indicates user input—text you type verbatim—in examples, such as <code>prompt> write_file top</code>
[]	Denotes optional arguments in syntax, such as <code>write_file [-format fmt]</code>
...	Indicates that arguments can be repeated as many times as needed, such as <code>pin1 pin2 ... pinN</code>
	Indicates a choice among alternatives, such as <code>low medium high</code>
Ctrl+C	Indicates a keyboard combination, such as holding down the Ctrl key and pressing C.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

Customer support is available through SolvNet online customer support and through contacting the Synopsys Technical Support Center.

Accessing SolvNet

SolvNet includes a knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services including software downloads, documentation, and technical support.

To access SolvNet, go to the following address:

<https://solvnet.synopsys.com>

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.

If you need help using SolvNet, click HELP in the top-right menu bar.

Contacting the Synopsys Technical Support Center

If you have problems, questions, or suggestions, you can contact the Synopsys Technical Support Center in the following ways:

- Open a support case to your local support center online by signing in to SolvNet at <https://solvnet.synopsys.com>, clicking Support, and then clicking “Open A Support Case.”
- Send an e-mail message to your local support center.
 - E-mail support_center@synopsys.com from within North America.
 - Find other local support center e-mail addresses at <http://www.synopsys.com/Support/GlobalSupportCenters/Pages>
- Telephone your local support center.
 - Call (800) 245-8005 from within North America.
 - Find other local support center telephone numbers at <http://www.synopsys.com/Support/GlobalSupportCenters/Pages>

1

Basic Concepts for Optimizing Designs

Optimizing (compiling) is the step in the synthesis process that attempts to implement a combination of library cells that meets the functional, speed, and area requirements of your design. Optimization transforms a design into a technology-specific circuit based on the attributes and constraints you place on the design.

The quality of optimization results depends on how the HDL description is written. In particular, the partitioning of the hierarchy in the HDL, if done well, can enhance optimization.

During optimization, the Design Compiler tool from Synopsys attempts to meet the constraints you have set on the design. Design Compiler's optimization algorithms use costs to determine if a design change is an improvement. Design Compiler calculates two cost functions: one for design rule constraints and one for optimization constraints and accepts an optimization move if it decreases the cost of one component without increasing more-important costs. By default, the design rule constraints (transition, fanout, capacitance, and cell degradation) have a higher priority than the optimization constraints (delay and area).

For basic information about optimization constraints and timing, see the *Design Compiler User Guide* and *Synopsys Timing Constraints and Optimization User Guide*.

This chapter contains the following sections:

- [Using DC Ultra](#)
- [Exploring the Design Space](#)

- [Optimization Phases](#)
- [Optimization Flow](#)

Using DC Ultra

The DC Ultra tool is applied to high-performance deep submicron ASIC and IC designs, where maximum control over the optimization process is required. In addition to the capabilities provided by DC-Expert, the DC Ultra tool provides the following features:

- Automatic area ungrouping
- Datapath optimization
- Finite state machine (FSM) optimization
- Advanced critical path resynthesis
- Register retiming
- Support for advanced cell modeling, that is, the cell-degradation design rule
- Advanced timing analysis

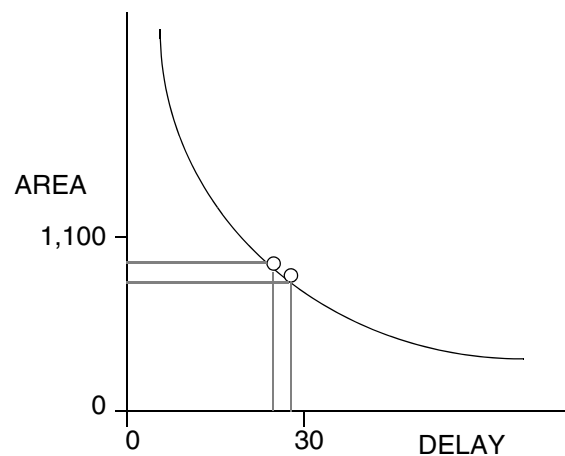
Features that are not available in DC-Expert are noted as such in the Design Compiler documentation.

To use any of these advanced optimization features, you use the `compile_ultra` command, which enables you to run a compile flow comprising of the most powerful features of DC Ultra, including topographical technology.

When you enter the `compile_ultra` command, Design Compiler checks to ensure that a DC Ultra license is available. If a DC Ultra license is not available, the tool issues a warning and uses the Design-Compiler license instead. If you use the `-force` option with the command and a DC Ultra license is not available, the return status is 0 (it is 1 otherwise) and compile stops.

Exploring the Design Space

Experimenting with speed and area to get the smallest or fastest design is called exploring the design space. Using Design Compiler, you can examine different implementations of the same design in a relatively short time. [Figure 1-1](#) shows a design space curve. The shape of the curve demonstrates the tradeoff between area-efficient and speed-efficient circuits.

Figure 1-1 Design Space Curve

Optimization Phases

The optimization process modifies the logic in a netlist. Optimization uses cells from the logic library in an attempt to meet specified constraints.

Design Compiler performs the following types of optimizations:

- Combinational optimization, including
 - Technology-independent optimization, which operates at the logic level. Design Compiler represents the gates as a set of Boolean logic equations.
 - Mapping, during which Design Compiler selects components from the logic library to implement the logic structure.
 - Technology-specific optimization, which operates at the gate level.
- Sequential optimization.
- Local optimizations.

Combinational Optimization

The combinational optimization phase transforms the logic-level description of the combinational logic to a gate-level netlist. [Figure 1-2](#) shows the logic-level description of the combinational logic and the gate-level optimization for design LED.

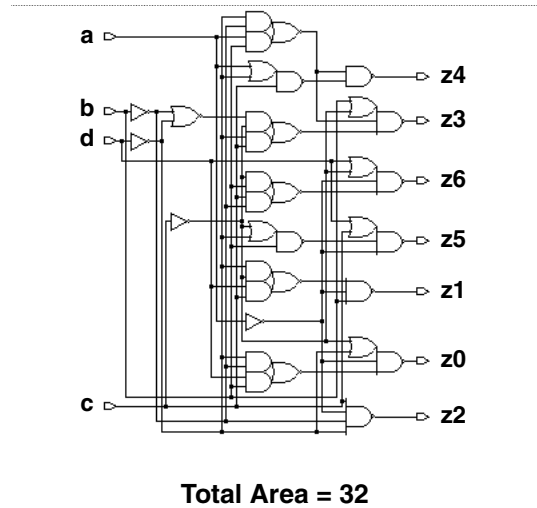
Figure 1-2 Logic-Level and Gate-Level Optimization for Design LED

```

design_name LED
.inputnames a b c d
.outputnames z0 z1 z2 z3 z4 z5 z6

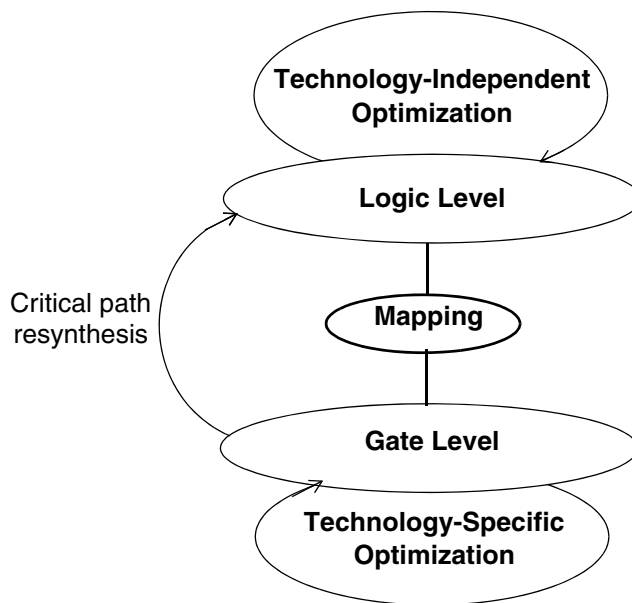
n1 = c' ;
n2 = d' ;
n3 = a' ;
n4 = ((n2' + n6') * (d' + b')) ;
n5 = ((n2' + n1') * (d' + c')) ;
n6 = b' ;
n7 = ((n2' + n6') * (a' + b')) ;
n8 = ((n12' + n1') * (n2' + c')) ;
n9 = ((a' * n2') + c') ;
n10 = ((n1' * n2') + b') ;
n11 = ((n1' + b') * (c' + n6')) ;
n12 = (n6' * n2') ;
z0 = ((n1' * n2') + n3' + n4') ;
z1 = (n5' + n3' + b') ;
z2 = (c' + n3' + n6' + n2') ;
z3 = ((b' * n1') + n7' + n8') ;
z4 = (n7' + n9') ;
z5 = ((d' * c') + n10' + n3') ;
z6 = ((d' * n1') + n3' + n11') ;

```



[Figure 1-3](#) shows the two levels of design representation and the related combinational optimization phases.

Figure 1-3 Combinational Optimization Phases

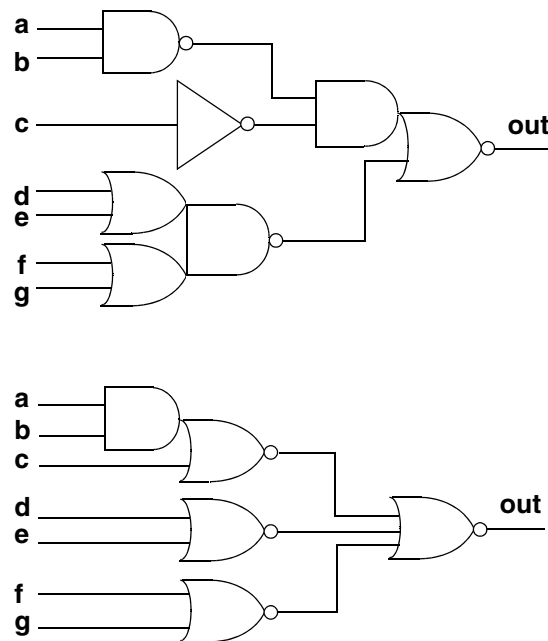


Technology-Independent Optimization

Technology-independent optimization applies algebraic and Boolean techniques to a set of logic equations. This optimization reimplements the logic equations to meet your timing and area goals.

Mapping

During mapping, Design Compiler selects components from the logic library to implement the logic structure. Design Compiler tries different logic combinations, using only components that approach the defined speed and area goals. [Figure 1-4](#) shows examples of mapped gates.

Figure 1-4 Mapped Gates

Technology-Specific Optimization

Technology-specific optimization synthesizes a gate-level design that attempts to meet your timing and area constraints.

Initial Sequential Optimization

Initial sequential optimization maps sequential cells to cells in the library. You can map to either standard sequential cells or scan-equivalent cells. Initial sequential optimization is in the first phase of gate-level optimization.

At this point in the optimization process, information about the delay through the combinational logic is incomplete. Design Compiler does not have enough information to select the optimum sequential cell. The tool can correct this lack of information later, in the final sequential optimization phase.

Final Sequential Optimization

Design Compiler has accurate values for all delays through the I/O pads and combinational logic before it enters the final sequential optimization phase. In this phase, Design Compiler optimizes timing-critical sequential cells (cells on the critical path).

The tool examines each sequential cell and its surrounding combinational logic to determine whether they might be replaced by more-optimal sequential cells from the target library in order to meet timing and area constraints.

Final sequential optimization can achieve the following:

- Improve design timing by choosing higher-performance sequential cells.
- Possibly to reduce area and delay of the design if more optimal sequential cells exist in the library. The tool incorporates the combinational logic in the sequential cell, as shown in [Figure 1-5](#) and [Figure 1-6](#).
- Further improve area by remapping sequential elements.

Figure 1-5 Sequential Optimization

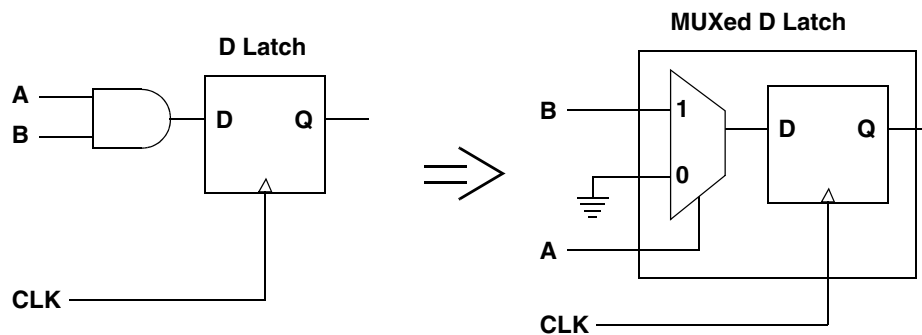
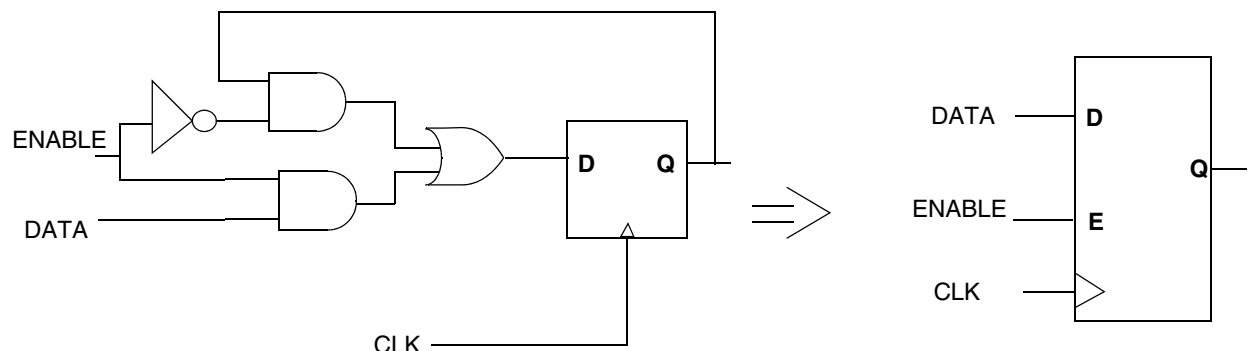


Figure 1-6 Sequential Optimization

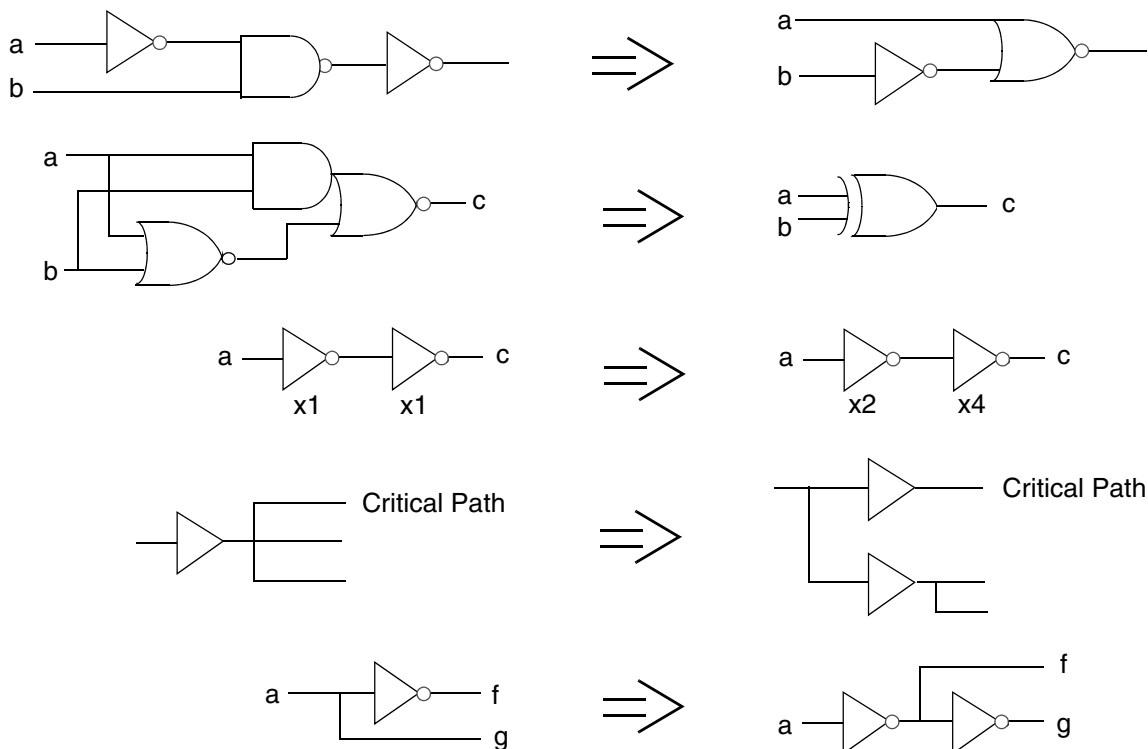


Local Optimizations

The final step in gate-level optimization involves making local changes. Design Compiler makes incremental modifications to the design to adjust for timing or area.

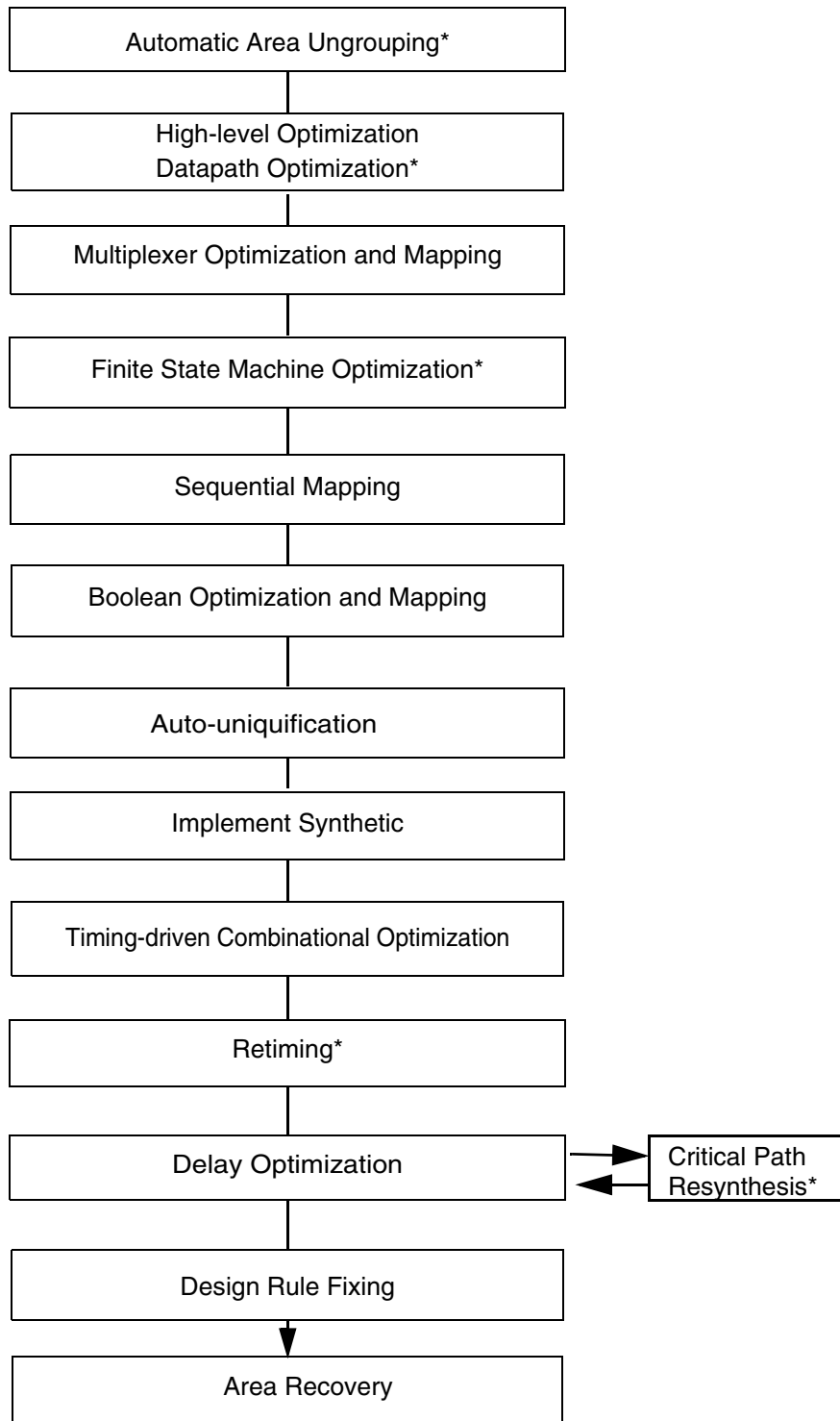
Figure 1-7 shows common local optimization steps.

Figure 1-7 Local Optimization Steps



Optimization Flow

Figure 1-8 on page 1-10 shows the optimization flow. Steps that occur only in DC Ultra are marked with an asterisk. The sections following the figure describe the steps in the flow.

Figure 1-8 Optimization Flow

Automatic Ungrouping

The `compile_ultra` command automatically ungroups logical hierarchies. Ungrouping merges subdesigns of a given level of the hierarchy into the parent cell or design. It removes hierarchical boundaries and allows Design Compiler to improve timing by reducing the levels of logic and to improve area by sharing logic. For information about the automatic ungrouping of hierarchies, see [“Automatic Ungrouping” on page 4-7](#).

You can also manually ungroup hierarchies by using the `ungroup` command or the `set_ungroup` command followed by `compile_ultra`. For information about manually ungrouping hierarchies, see “Removing Levels of Hierarchy” in the *Design Compiler User Guide*.

High-Level Optimization and Datapath Optimization

During high-level optimization, resources are allocated and shared, depending on timing and area considerations. Resource sharing enables the tool to build one hardware component for multiple operations, which typically reduces the hardware required to implement your design. Additional optimizations, such as arithmetic optimization and the sharing of common subexpressions, are also performed. In addition, if you are using the DC Ultra tool, advanced datapath transformations are performed. For more information, see [Chapter 5, “High-Level Optimization and Datapath Optimization.”](#)

Multiplexer Mapping and Optimization

In this phase, Design Compiler maps combinational logic representing multiplexers in the HDL code directly to a single multiplexer (MUX) or a tree of multiplexer cells from the target logic library. Design Compiler reorders SELECT signals for better area and shares multiplexer trees. For more information, see [Chapter 6, “Multiplexer Mapping and Optimization.”](#)

Finite State Machine Optimization

DC Ultra can perform automatic extraction and optimization of finite state machines (FSMs) from your RTL. If an FSM subdesign is represented as a state table, it is read directly and optimized. If the FSM subdesign is represented in an HDL format or as part of an input netlist, the FSM is extracted first and then optimized. The FSM part of the optimization includes optimization with respect to state assignment and, optionally, state minimization. After FSM optimization, the FSM subdesign is converted to Boolean equations and technology-independent flip-flops and compiled along with the rest of the design, using standard optimization procedures.

FSM optimization is carried out automatically. For more information, see [Chapter 8, “Optimizing Finite State Machines.”](#)

Sequential Mapping

This phase consists of two steps: register inferencing by HDL Compiler and technology mapping by Design Compiler. The RTL is translated into a technology-independent representation called SEQGEN; the SEQGEN is then mapped to gates from the logic library. A SEQGEN is a generic sequential element that is used by Synopsys tools to represent registers and latches in a design. SEQGENs are created during elaboration and are mapped to flip-flops or latches during compile. For more information, see [Chapter 8, “Optimizing Finite State Machines.”](#)

Structuring and Mapping

In this phase, the tool optimizes unmapped unstructured logic and maps it to technology gates. Structuring is an optimization step that adds intermediate variables and logic structure to a design. During structuring, Design Compiler searches for subfunctions that can be factored out, then evaluates these factors based on the size of the factor and the number of times the factor appears in the design. The subfunctions that most reduce the logic are turned into intermediate variables and factored out of the design equations. Design Compiler offers timing-driven structuring to minimize delays and Boolean structuring to reduce area.

Auto-Uniquification

The uniquify process copies and renames any multiply referenced design so that each instance references a unique design. The process removes the original design from memory after it creates the new, unique designs. The original design and any collections that contain it or its objects are no longer accessible. In earlier releases, you had manually to run the `uniquify` command to create a uniquely named copy of the design for each instance. The tool automatically uniquifies designs as part of the compile process. For more information on the uniquification process, see the *Design Compiler User Guide*.

Implementing Synthetic Parts

HDL operators (either built-in operators like `+` and `*` or HDL functions and procedures) are associated with synthetic parts, which are bound in turn to synthetic modules. Each synthetic module can have multiple architectural realizations, called implementations.

For example, when you use the HDL addition operator (+), HDL Compiler infers the need for an adder resource and puts an abstract representation of the addition operation in the netlist. During high-level optimization, the tool manipulates this abstract representation—called a synthetic operator—and applies optimizations such as arithmetic optimization or resource sharing.

During the high-level optimization phase, the tool used abstract representations for synthetic parts. During the implement synthetic parts phase, the tool maps synthetic modules to architectural representations (implementations). For more information on synthetic parts, see the HDL Compiler documentation and the DesignWare documentation.

Timing-Driven Combinational Optimization

During this phase, Design Compiler performs optimization of combinational parts. This phase has two components: timing-driven structuring and incremental implementation selection. During timing-driven structuring, the tool restructures logic in the critical paths to improve delay cost. During incremental implementation selection, the tool explores alternative implementations for each synthetic operator. It evaluates and replaces synthetic implementations along the critical path to improve delay cost.

Register Retiming

Register retiming is available in DC Ultra only. Register retiming is a sequential optimization technique that moves registers through the combinational logic gates of a design to optimize timing and area. Register retiming adds an opportunity for improving circuit timing.

Design Compiler provides two ways to perform register retiming:

- The `optimize_registers` command performs retiming of sequential cells (edge-triggered registers or level-sensitive latches) for pipelined designs. For information on the `optimize_registers` command, see the *Design Compiler Register Retiming Reference Manual*.
- The `compile_ultra` command supports the `-retime` option, which enables Design Compiler to automatically perform local retiming moves to improve worst negative slack (WNS). This capability, called adaptive retiming, optimizes an entire design. It works best with general non-pipelined logic.

For information on adaptive retiming, see [Chapter 9, “Sequential Mapping.”](#)

Delay Optimization

In this phase, Design Compiler attempts to fix existing delay violations by traversing the critical path. During delay optimization, Design Compiler reevaluates existing

implementations to determine if they meet constraints. If constraints are not met, Design Compiler selects a different implementation. It applies local transformations such as upsizing, load isolation and splitting, and revisits mapping of sequential paths. [Figure 1-7](#) shows some common local optimization steps. Design Compiler takes design rules into account during this phase. When two circuit solutions offer the same delay performance, Design Compiler implements the solution that has the lower design rule cost.

In addition, Design Compiler performs critical path resynthesis on your design to improve timing. It identifies the critical path and attempts to do a full compile on only the logic along that path. This process then repeats on the new critical path.

The `compile` command, with its `-map_effort high` option, enables critical path resynthesis. For more information, see [Chapter 10, “Adaptive Retiming.”](#)

Note:

Critical path resynthesis requires a DC Ultra license.

Design Rule Fixing

Design rules are provided in the vendor logic library to ensure that the product meets specifications and works as intended. In this phase, Design Compiler fixes any design rule violations. Whenever possible, Design Compiler fixes design rule violations by resizing gates across multiple logic levels—as opposed to adding buffers to the circuitry. For more information, see [Chapter 10, “Adaptive Retiming.”](#)

Area Optimization

Assuming that you have placed area constraints on your design (with the `set_max_area` command), Design Compiler now attempts to minimize the number of gates in the design. Using the `-map_effort` or `-area_effort` option of the `compile` command, you can direct Design Compiler to put a low, medium, or high effort into area optimization. (If you do not place area constraints on your design, Design Compiler performs a limited series of downsizing and area cleanup steps.) For more information, see [Chapter 10, “Adaptive Retiming.”](#)

2

Compiling a Design

When you compile a design, Design Compiler attempts to implement a combination of library cells that meets the functional, speed, and area requirements of a design according to the attributes and constraints placed on the design. You use the `compile_ultra` command or the `compile` command to compile a design. Design Compiler provides options that enable you to customize and control optimization. The optimization process trades off timing and area constraints to provide the smallest possible circuit that meets the specified timing requirements.

During compile, Design Compiler's optimization algorithms use costs to determine if a design change is an improvement. Design Compiler calculates two cost functions: one for design rule constraints and one for optimization constraints and accepts an optimization move if it decreases the cost of one component without increasing more-important costs. For more information on the compile cost function, see [Chapter 10, "Adaptive Retiming."](#)

During a full compile, Design Compiler removes the existing gate structure from a design, then rebuilds the design. A full compile performs both technology-independent optimization as well as technology-specific optimization (mapping).

In addition, different pieces of your design require different compilation strategies, such as a top-down hierarchical compile or bottom-up compile. You need to develop a compilation strategy before you compile. You can use various strategies to compile, depending on your design, and you can mix strategies.

This chapter contains the following sections:

- [Before You Start](#)
- [The compile_ultra Command](#)
- [The compile Command](#)
- [Performing High-Effort Synthesis](#)
- [Performing an Incremental Compile](#)
- [Performing Test-Ready Compile](#)
- [Controlling Design Rule Fixing](#)
- [Performing a Top-Level Compile](#)
- [Choosing a Compile Strategy](#)
- [Redirecting the Output of Commands](#)
- [Checking the Compile Log](#)

Before You Start

To optimize successfully, do the following before you compile a design:

- Ensure that partitioning is the best possible for synthesis. The quality of optimization results depends on how the HDL description is written. In particular, the partitioning of the hierarchy in the HDL, if done well, can enhance optimization.
- Define constraints as accurately as possible but do not overconstrain your design.
- Use a good synthesis library.
- Identify all multicycle and false paths.
- Instantiate clock gating elements.
- Optionally, define scan style.
- Determine the best strategy for optimizing your design.

The `compile_ultra` Command

For designs that have significantly tight timing constraints, you can invoke a single DC Ultra command, `compile_ultra`, for better quality of results (QoR). The command is a push-button solution for timing-critical, high performance designs and encapsulates DC Ultra strategies into a single command. It enables you to apply the best possible set of timing-centric variables or commands during compile for critical delay optimization as well as improvement in area QoR. Because the `compile_ultra` command includes all compile options and starts the entire compile process, no separate `compile` command is necessary.

To use the `compile_ultra` command, you need a DC Ultra license and a DesignWare Foundation license.

The DesignWare library is necessary so that the tool can use licensed DesignWare architectures for optimal QoR. By default, if the `dw_foundation.sldb` library is not in the synthetic library list but the DesignWare license has been successfully checked out, the `dw_foundation.sldb` library is automatically added to the synthetic library list. This behavior applies to the current command only. The user-specified synthetic library and link library lists are not affected.

In addition, all DesignWare hierarchies are, by default, unconditionally ungrouped in the second pass of the compile. You can prevent this ungrouping by setting the `compile_ultra_ungroup_dw` variable to false (the default is true).

The `compile_ultra` command includes the following features:

- Topographical technology, which enables you to accurately predict post-layout timing, area, and power during RTL synthesis without the need for wireload model-based timing approximations. Topographical mode uses Synopsys' placement and optimization technologies to drive accurate timing prediction within synthesis, ensuring better correlation to the final physical design. For more information, see the *Design Compiler User Guide*.

- Automatic boundary optimization

By default, the `compile_ultra` command optimizes across hierarchical boundaries. Boundary optimization is a strategy that can improve a hierarchical design by allowing the compile process to modify the port interface of lower-level designs. Use the `-no_boundary_optimization` option to turn off boundary optimization. For more information on boundary optimization, see [Chapter 3, "Optimization Techniques."](#)

- Automatic ungrouping

The `compile_ultra` command automatically ungroups logical hierarchies. Ungrouping merges subdesigns of a given level of the hierarchy into the parent cell or design. It removes hierarchical boundaries and allows Design Compiler to improve timing by reducing the levels of logic and to improve area by sharing logic. For information about the automatic ungrouping of hierarchies, see ["Automatic Ungrouping" on page 4-7](#).

- Aggressive logic duplication for load isolation

The tool considers a larger section of the critical path and replicates many gates to isolate the load observed by the critical path. For more information, see [Chapter 3, "Optimization Techniques."](#)

- Library-aware mapping and structuring

This capability enables you to characterize your target logic library and create a pseudolibrary called ALIB, which has mappings from Boolean functional circuits to actual gates from the target library. The ALIB file provides Design Compiler with greater flexibility and a larger solution space to explore tradeoffs between area and delay during optimization. For more information, see the *Design Compiler User Guide*.

- Automatic datapath extraction and optimization

Datapath extraction transforms arithmetic operators (for example, addition, subtraction, and multiplication) into datapath blocks. During datapath implementation, the tool uses a datapath generator to generate the best implementations for these extracted components. For more information, see [Chapter 5, "High-Level Optimization and Datapath Optimization."](#)

Note:

Compile options, such as `-map_effort` and `-area_effort`, are not compatible with the `compile_ultra` command.

The compile Command

When the current design is hierarchical and there are multiple instances of a subdesign, it is no longer necessary to run the `uniquify` command before running the `compile` command. The tool automatically uniquifies designs as part of the compilation process. Use the `check_design -multiple_designs` command to report information messages related to multiply-instantiated designs. (By default, these messages are suppressed.)

Controlling Mapping Optimization

Select appropriate effort levels for mapping optimization by using the `compile` command with the `-map_effort` and `-area_effort` options. [Table 2-1](#) lists the arguments you can use with the `-map_effort` option.

Table 2-1 Mapping Optimization Effort Levels

Argument	Description
<code>-map_effort medium</code>	This option is the default. Design Compiler tries to find a good mapping but does not use some CPU-intensive strategies. Medium is appropriate for getting a quick idea of how large a circuit will be. Use medium for most cases.
<code>-map_effort high</code>	This option performs all medium effort optimizations and critical path resynthesis. It takes significantly longer to compile but can produce better designs. The mapping process proceeds until it has tried all strategies.

Note:

The DC Ultra version of the `compile` command `-map_effort high` option includes algorithms that enable mapping to wide-fanin gates. This reduces critical path length.

Customizing the compile Command

Compile variables enable you to customize the `compile` command to fit your particular needs. Set variables from the `dc_shell` prompt, or define them in your `.synopsys_dc.setup` file.

The `compile_variables` man page lists the compile variables.

The syntax is

```
set variable_name value
```

The individual variable descriptions define the value type.

To list compile variables and their current values, enter

```
dc_shell> print_variable_group compile
```

Performing High-Effort Synthesis

The `compile_ultra` command supports two options, which encapsulate higher-effort synthesis strategies. The `-area_high_effort_script` option and `-timing_high_effort_script` option include encapsulated scripts that offer additional area and timing improvements. Depending on the optimization goal, the scripts apply a compile strategy that might turn on or off different optimization features.

Performing an Incremental Compile

An incremental compile might improve quality of results (QoR) by improving the structure of your design after the initial compile. To perform an incremental compile, use the `-incremental` option of the `compile_ultra` command or the `-incremental_mapping` option of the `compile` command.

Incremental mapping uses the existing gates from an earlier compilation as a starting point for the mapping process. It improves the existing design cost by focusing on the areas of the design that do not meet constraints and affects the second pass of compile. The existing structure is preserved if all constraints are already met. Mapping optimizations are accepted only if they improve the circuit speed or area.

Keep the following points in mind when you do an incremental compile:

- The option enables only gate-level optimizations. For more information, see [Chapter 10, “Adaptive Retiming.”](#)
- Gates are not converted back to the generic technology (GTECH) level.
- Flattening and structuring are not done on the mapped portion of the design.
- Implementations for DesignWare operators are reselected if optimization costs can be improved.

The `compile_ultra -incremental` command is incompatible with the following options:

```
-top
-timing_high_effort_script
-area_high_effort_script
```

Performing Test-Ready Compile

Test-ready compile reduces iterations and design time, by accounting for the impact of the scan implementation during the logic optimization process. The optimization cost functions consider the impact of the scan cells themselves and the additional loading due to the scan-chain routing. By accounting for the timing impact of scan design from the start of the synthesis process, test-ready compilation eliminates the need to recompile your design after scan insertion. Use the `-scan` option of the `compile_ultra` command or the `compile` command to enable test-ready compile. When you use this option, the tool replaces all sequential elements during optimization.

For more information, see the [“Performing Test-Ready Compile” on page 9-20](#).

Controlling Design Rule Fixing

You can direct Design Compiler to avoid design rule fixing or to compile with only design rule fixing. The `-no_design_rule` and `-only_design_rule` options of the `compile_ultra` command or `compile` command determine whether design rule violations are fixed before compilation stops. You cannot use these options together.

Table 2-2 Using Design Rule Fixing Options

Argument	Description
<code>-no_design_rule</code>	Causes compile to exit before fixing design rule violations. This allows you to check the results in a constraint report before fixing the violations.
<code>-only_design_rule</code>	Causes compile to perform only design rule fixing. Mapping optimizations are not performed. If you are using the <code>compile_ultra</code> command, you must use the <code>-only_design_rule</code> option with the <code>-incremental</code> option.

If you omit the `-no_design_rule` and `-only_design_rule` options, Design Compiler performs both design rule fixing and mapping optimizations before exiting (the default). The `compile` command supports an additional option, `-only_hold_time`, which causes the

`compile` command to perform only hold time fixing, ignoring other design rules. The `set_fix_hold` command must be specified for hold time fixing to be performed. For more information on design rule fixing, see [“Design Rule Fixing” on page 11-14](#).

Note:

Another way to limit design rule fixing is by using the `set_cost_priority` command with the `-delay` option.

Performing a Top-Level Compile

The `-top` option of the `compile_ultra` command or the `compile` command invokes the top-level optimization process. The top-level optimization capability fixes constraint violations occurring at the top level after the subblocks in a design are assembled. These violations might occur due to changes in the environment around the subblocks as a result of the optimizations that have been performed in the subblocks.

Top-level optimization fixes only violations of top-level nets because it is assumed that the subblocks have been compiled separately and are meeting timing. However, any design rule violations present in the design will be fixed regardless of where the violation occurs.

The `-top` option works with mapped designs only and runs significantly faster than an incremental compilation because of its emphasis on top-level nets. Additionally, it does not perform any incremental implementation selection of synthetic components, structuring, or area recovery on the design.

Using the -top Option With Other Compile Options

The `compile_ultra -top` option is incompatible with the following options:

- `-incremental`
- `-timing_high_effort_script`
- `-area_high_effort_script`

The `compile -top` option is incompatible with the following options:

- `-incremental_mapping`
- `-exact_map`
- `-no_map`
- `-area_effort`

Limiting Optimization to Paths Within a Specific Range

By default, the `-top` option aims to fix all design rule violations in the entire design but only the intermodule timing paths with violations. It does not address the intramodule timing paths. To direct Design Compiler to attempt to optimize the intermodule paths with timing delay violations within a specific range, set the `critical_range` attribute before you compile with the `-top` option.

Fixing Timing Violations For All Paths

By default, when the `-top` option is used, Design Compiler fixes all design rules but only those timing violations whose paths cross top-level hierarchical boundaries.

Setting the `compile_top_all_paths` environment variable to true causes the `-top` option to attempt to fix timing violations for all paths.

Choosing a Compile Strategy

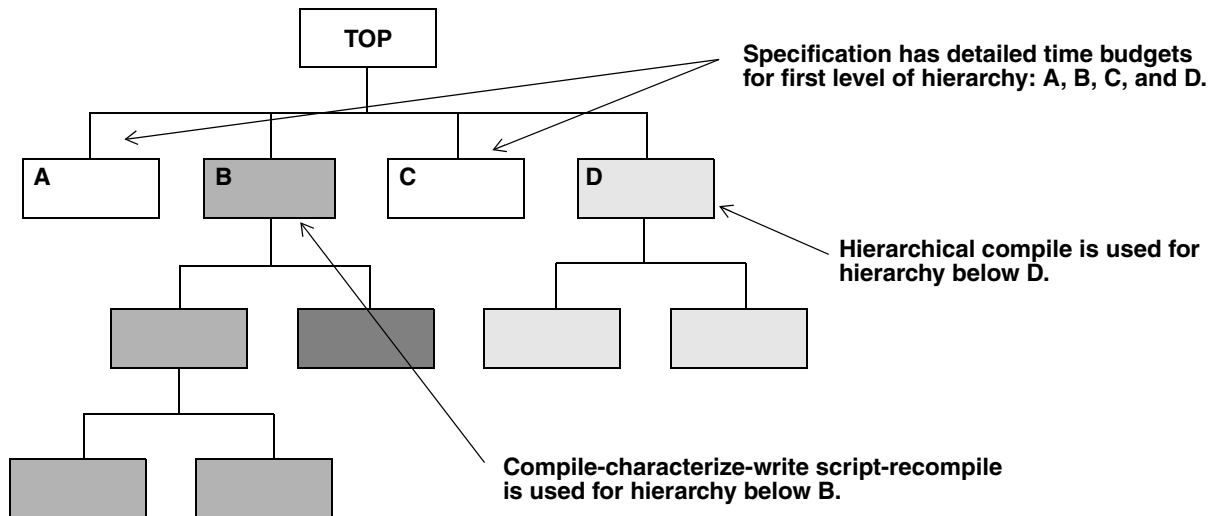
The two strategies for compiling a hierarchical design are

- Top-down hierarchical compile (recommended)
- Compile-characterize-write script-recompile. This strategy is also known as the bottom-up compile strategy.

Mixing Compilation Strategies

You can mix the two compilation strategies, as shown in [Figure 2-1](#).

Figure 2-1 Mixing Compilation Strategies



Using the Top-Down Hierarchical Compile Strategy

Design Compiler automatically compiles hierarchical circuits without collapsing the hierarchy. After each module in the design is compiled, Design Compiler continues to optimize the circuit until the constraints are met. This process sometimes requires recompiling subdesigns on a critical path. When the performance goals are achieved or when no further improvement can be made, the compile process stops.

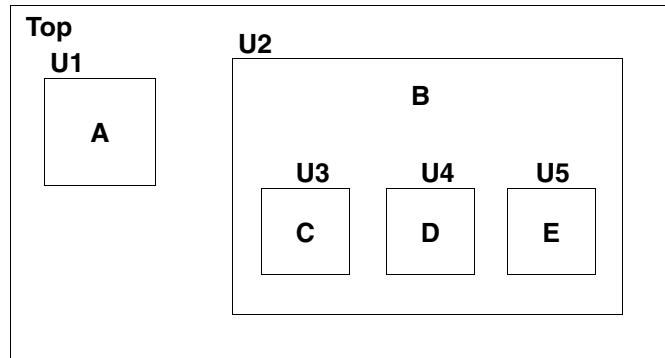
Hierarchical compilation is automatic when the design being compiled has multiple levels of hierarchy that are not marked `dont_touch`. Design Compiler preserves the hierarchy information and optimizes individual levels automatically, based on the constraints at the top level of hierarchy (`dont_touch` attributes placed on the top level of hierarchy are ignored).

The top-down hierarchical compile strategy is an easy, push-button approach that involves only three basic steps:

1. Read in the entire design.
2. Apply constraints and attributes to the top level. Constraints and attributes are based on the design specification.
3. Compile.

Example

Use top-down hierarchical compile for design Top.



The design specification for design Top is

Operating conditions:WCCOM

Wire Load Model: "20x20"

clock: 22 MHz

Input delay time: 10 ns

Use the compile script shown in [Example 2-1](#) to run top-down hierarchical compile for design Top. The script includes a script of default constraints, default.con.

Example 2-1 Top-Down Hierarchical Compile Script

```

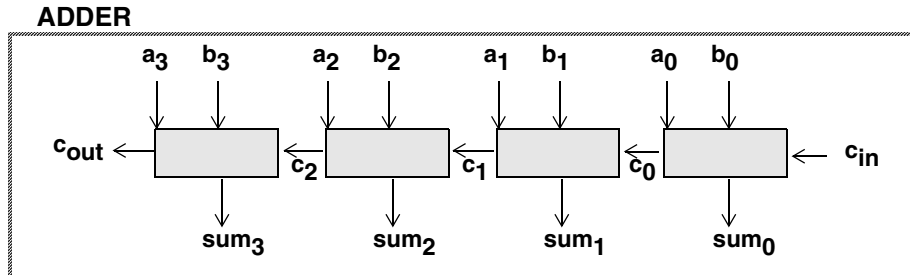
read_file -format vhdl top.vhd
current_design Top
source default.con
set_operating_conditions WCCOM
set_wire_load_model -name "20x20"
create_clock -period 45 clk
set_input_delay 10 -clock clk [all_inputs]
set_output_delay
compile_ultra
  
```

Running a Top-Down Hierarchical Compile Strategy

This example of running the hierarchical optimization strategy uses a hierarchical, 4-bit adder design named ADDER. ADDER is composed of four 1-bit full adders with A, B, and CIN (carry in) inputs and SUM and COUT (carry out) outputs.

Figure 2-2 shows a diagram of design ADDER.

Figure 2-2 Hierarchical Design



The suggested optimization strategy for hierarchical designs is

1. Read in as much of the design hierarchy as you need to use in this session. For example, enter

```
dc_shell> read_file -format ddc adder.ddc
Reading ddc file '/usr/design/adder.ddc'
Current design is 'ADDER'.
```

2. Run the `check_design -multiple_designs` command to determine whether any subdesigns (components) are referenced more than once (unresolved) or whether the hierarchy is recursive. The command reports all multiply instantiated designs along with instance names and associated attributes (`dont_touch`, `black_box`, and `ungroup`).

For example, enter

```
dc_shell> check_design -multiple_designs
Information: Design 'FULL_ADDER' is instantiated 4 times.
          Cell 'ADD0' in design 'ADDER'
          Cell 'ADD1' in design 'ADDER'
          Cell 'ADD2' in design 'ADDER'
          Cell 'ADD3' in design 'ADDER'
```

3. Resolve multiple instances.

For each cell referencing a nonunique subdesign, you can let compile automatically unquify the multiple instances, or, before running compile, you can do one of the following actions (you can use each action with one or more cells):

- Combine the cell into the surrounding circuitry (`ungroup`).
- Compile the cell separately, then use `set_dont_touch`.

Note:

The tool automatically unquifies designs as part of the compile process.

Note that you can still manually force the tool to uniquify designs before compile by running the `uniquify` command, but this step contributes to longer runtime because the tool automatically “re-uniquifies” the designs when you compile the design. You cannot turn off the uniquify process. For more information about resolving multiple instances, see the *Design Compiler User Guide*.

4. Check the design again to verify that all multiple instances have been uniquified, or ungrouped or have the `dont_touch` attribute set. For example, enter

```
dc_shell> check_design -multiple_designs
```

5. Compile the design.

```
dc_shell> compile_ultra
```

Using the Compile-Characterize-Write Script-Recompile Strategy

The compile-characterize-write script-recompile strategy is an alternative to hierarchical compilation. Using this strategy, first optimize nonunique designs, using context information or time budgets. Then, optimize higher-level blocks with the lower blocks marked as `dont_touch`.

Use the compile-characterize-write script-recompile strategy for medium and large designs that do not have good interblock specifications (the typical situation for many designs).

The compile-characterize-write script-recompile strategy assists in compiling large designs, using the divide-and-conquer approach, and is not limited by memory.

The compile-characterize-write script-recompile strategy has these disadvantages:

- It requires iterations until the interfaces are stable.
- It requires manual revision control.

The compile-characterize-write script-recompile strategy requires seven steps:

1. Compile subblocks independently, using estimates for drive and load. Use a default script to estimate drive and load.
2. Read in the entire compiled design.
3. Characterize one subblock.
4. Use the `write_script` command to save the information from the characterization.
5. Clear memory; read in the previously characterized subblock; and recompile the subblock, using the saved script.

For characterization information to apply, you must read in the database format that was characterized (.ddc format).

6. Read in the entire compiled design again without the old subblock; use recompiled subblock.
7. Choose another subblock, and repeat steps 3 through 7 until all subblocks are recompiled, using their actual environments.

When you use the compile-characterize-write script-recompile strategy, consider the following:

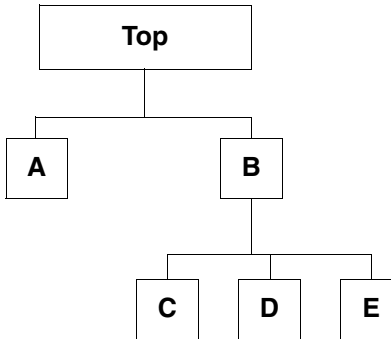
- If you do not modify your RTL code after the first time you read it in, you can save it to a .ddc file. This step saves time later when you reread the design.
- `compile` automatically goes to the submodule. If you want the compile to affect only the current design, you can either
 - Remove or omit the submodule from your database or
 - Use the `set_dont_touch` command to set the `dont_touch` attribute on the submodule.
- `compile` is bottom up.
- `characterize` is top down.
- By default, compile modifies the original copy in the current design. This could be a problem when the same design is referenced from multiple modules and these modules are compiled separately in sequence. For example, the first compile could change the interface or the functionality of the design by boundary optimization. When this design is referenced from another module in the subsequent compile, the modified design is uniquified and used.

You can set the `compile_keep_original_for_external_references` variable to true, which enables compile to keep the original design when there is an external reference to the design. When the variable is set to true, the original design and its sub-designs are copied and preserved (before doing any modifications during compile) if there is an external reference to this design.

Typically, you require this variable only when you are doing a bottom-up compile without setting a `dont_touch` attribute on all the sub-designs, especially those with boundary optimizations turned on. If there is a `dont_touch` attribute on any of the instances of the design or in the design, this variable has no effect.

Example

Use the compile-characterize-write script-recompile compilation strategy for design Top.



The compile script shown in [Example 2-2](#) runs compile-characterize-write script-recompile for design Top. [Example 2-3](#) shows the default constraint script used to estimate the drive and the load, as shown in [Figure 2-3](#).

Example 2-2 compile-characterize-write script-recompile Script

```

read_ddc {top.ddc B.ddc ...}
source defaults.con
characterize B.blk
current_design B
write_script > B.wscr
remove_design -all
read_ddc B.ddc
source B.wtcl
compile
read_ddc {top.ddc A.ddc}
source defaults.con
characterize A.blk
current_design A
write_script > A.wscr

```

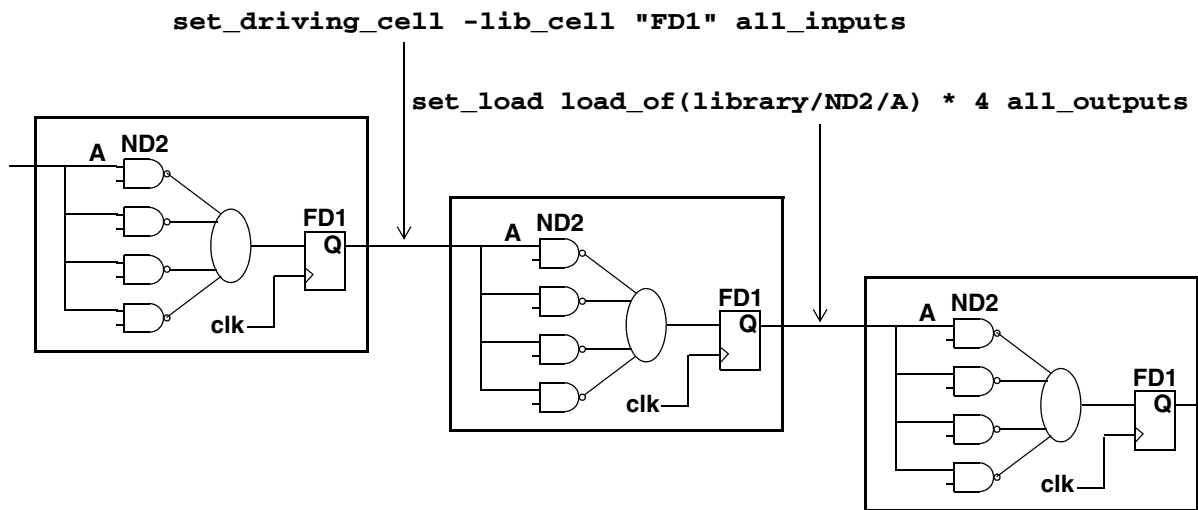
Example 2-3 Default Constraint Script (default.con)

```

set_operating_conditions "WCCOM"
set_wire_load_model -name "10x10"
create_clock -period 20 clk
dont_touch_network clk
set_input_delay -clock clk 4 [all_inputs]
set_output_delay -clock clk 5 [all_outputs]
set_load load_of(library/ND2/A) * 4 [all_outputs]
set_driving_cell -lib_cell "FD1" [all_inputs]
set_drive 0 clk

```

Figure 2-3 Estimating Drive and Load



Redirecting the Output of Commands

You can redirect or append the output of the commands to a file you can review. This way, you can archive runtime messages for future reference.

Table 2-3 Redirecting the Command Output

To do this	Use this
Divert command output to a file.	> (redirection operator)
Append command output to a file.	>> (append operator)
Redirect command output to a file.	redirect command

Note:

The pipe character (|) has no meaning in the dc_shell interface.

Checking the Compile Log

During optimization, each time Design Compiler starts a step it prints a message in the compile log, indicating its progress.

For example

```
dc_shell> compile
```

```
Beginning Mapping Optimizations (Medium effort)
```

Trials	Area	Delta delay	Total neg slack	Design rule cost
3	1296477.2	7.58	3468.1	2.9
1	1296538.9	7.48	3382.7	2.9

For additional information, see [“Compile Log” on page 11-10](#) and the chapter on analyzing and resolving design problems in the *Design Compiler User Guide*.

3

Optimization Techniques

Optimizing a design produces the smallest design that meets your constraints. The `compile_ultra` and `compile` commands enable optimization. The optimization process trades off timing and area constraints to provide the smallest possible circuit that meets the specified timing requirements.

This chapter has the following sections:

- [Optimizing for Delay](#)
- [Optimizing for Area](#)
- [Creating Path Groups](#)
- [Optimizing Near-Critical Paths](#)
- [Fixing Heavily Loaded Nets](#)
- [Performing High-Effort Compile](#)
- [Performing a High-Effort Incremental Compile](#)
- [Disabling Total Negative Slack Optimization](#)
- [Optimizing Across Hierarchical Boundaries](#)
- [Isolating Input and Output Ports](#)
- [Propagating Constants](#)
- [Enabling Critical Path Resynthesis](#)

- [Logic Duplication and Mapping to Wide-Fanin Gates](#)
- [Removing Hierarchy](#)
- [Optimizing for Multiple Clocks Per Register](#)
- [Preserving the Clock Network After Clock Tree Synthesis](#)
- [Optimizing Once for Best- and Worst-Case Conditions](#)
- [Optimizing With Multiple Libraries](#)
- [Fixing Nets Connected to Multiple Ports](#)
- [Building a Balanced Buffer Tree](#)
- [Defining a Signal for Unattached Master Clocks](#)

Optimizing for Delay

Design Compiler optimizes the timing of your design based on the delay constraints you specify. Constraints affecting delay include clocks, input and output delays, external loads, input driving cells, operating conditions, and wire load tables.

The following strategies can help achieve a faster design:

- Automatically ungroup hierarchies along the critical path by using the `compile_ultra` command. See [“Automatic Ungrouping” on page 4-7](#).
- Ungroup all or part of the hierarchy. This can give optimization more freedom to change logic that previously spanned a hierarchical boundary. See [“Removing Hierarchy” on page 3-13](#).
- Enable boundary optimization. See [“Optimizing Across Hierarchical Boundaries” in Chapter 4](#).
- Use the `group_path` command to create path groups. See [“Creating Path Groups” on page 3-4](#).
- Specify a critical range, so that Design Compiler optimizes not only the critical path but paths within that range of the critical path as well. See [“Optimizing Near-Critical Paths” on page 3-5](#).
- In the design exploration phase, you might want to give delay a higher priority than design rules, using the `set_cost_priority -delay` command.
- Use the `-timing_high_effort_script` option of the `compile_ultra` command. This option includes several strategies and settings for timing improvements.
- If you are using the `compile` command, perform a high-effort compile or a high-effort incremental compile. See [“Performing High-Effort Compile” on page 3-7](#) and [“Performing a High-Effort Incremental Compile” on page 3-8](#).
- Fix heavily loaded nets by using the `balance_buffer` command or by fixing design rules. See [“Fixing Heavily Loaded Nets” on page 3-6](#).

Optimizing for Area

The area optimization process minimizes the area your design uses. The process tries to improve area, if that can be done without degrading delay cost.

Area optimization requires that you set an area constraint, using the `set_max_area` command.

By default, area optimization does not create a new timing violation or worsen an existing timing violation to gain an improvement in area. If Design Compiler can identify a change to a path with a timing violation, it makes the change only if it can improve area without worsening the timing violation.

Achieving the smallest design can require changes in optimization strategy or rewriting the HDL code. Some strategies that can help achieve small designs are:

- Ungroup all or part of the hierarchy. Take care not to ungroup regular structures such as adders. In particular, ungroup smaller blocks to allow shared optimization across boundaries. See [“Removing Hierarchy” on page 3-13](#).
- Automatically ungroup small hierarchies by using the `compile_ultra` command or the `-auto_ungroup area` option of the `compile` command. See [“Automatic Ungrouping” in Chapter 4](#).
- Optimize across hierarchical boundaries. See [“Optimizing Across Hierarchical Boundaries” in Chapter 4](#).
- Disable total negative slack optimization by using the `-ignore_tns` option of the `set_max_area` command. See [“Disabling Total Negative Slack Optimization” on page 3-8](#).
- Use the `-area_effort high` option or `map_effort high` option of the `compile` command.
- Use the `-area_high_effort_script` option of the `compile_ultra` command.

Creating Path Groups

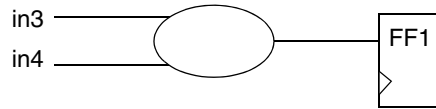
By default, Design Compiler groups paths based on the clock controlling the endpoint (all paths not associated with a clock are in the default path group). If your design has complex clocking, complex timing requirements, or complex constraints, you can create path groups to focus Design Compiler on specific critical paths in your design.

You can control the optimization of your design by creating and prioritizing path groups, which affect only the maximum delay cost function. By default, Design Compiler works only on the worst violator in each group.

Set the path group priorities by assigning weights to each group (the default weight is 1.0). The weight can be from 0.0 to 100.0.

For example, [Figure 3-1](#) shows a design that has multiple paths to flip-flop FF1.

Figure 3-1 Path Group Example



To indicate that the path from input in3 to FF1 is the highest-priority path, use the following command to create a high-priority path group:

```
dc_shell> group_path -name group3 -from in3 -to FF1/D -weight 2.5
```

Optimizing Near-Critical Paths

When you add a critical range to a path group, you change the maximum delay cost function from worst negative slack to critical negative slack. Design Compiler optimizes all paths within the critical range.

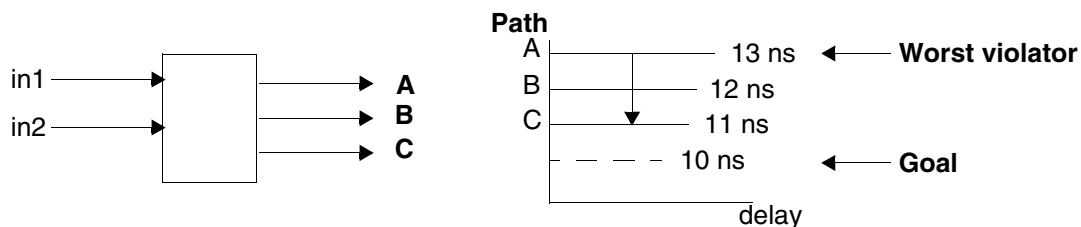
Specifying a critical range can increase runtime. To limit the runtime increase, use critical range only during the final implementation phase of the design, and use a reasonable critical range value. A guideline for the maximum critical range value is 10 percent of the clock period.

Use one of the following methods to specify the critical range:

- Use the `-critical_range` option of the `group_path` command.
- Use the `set_critical_range` command.

For example, [Figure 3-2](#) shows a design with three outputs, A, B, and C.

Figure 3-2 Critical Range Example



Assume that the clock period is 20 ns, the maximum delay on each of these outputs is 10 ns, and the path delays are as shown. By default, Design Compiler optimizes only the worst

violator (the path to output A). To optimize all paths, set the critical delay to 3.0 ns. For example,

```
create_clock -period 20 clk
set_critical_range 3.0 $current_design
set_max_delay 10 {A B C}
group_path -name group1 -to {A B C}
```

Fixing Heavily Loaded Nets

Heavily loaded nets often become critical paths. To reduce the load on a net, you can use either of two approaches:

- If the large load resides in a single module and the module contains no hierarchy, fix the heavily loaded net by using the `balance_buffer` command. For example, enter

```
source constraints.con
compile_ultra
balance_buffer -from [get_pins buf1/Z]
```

Note:

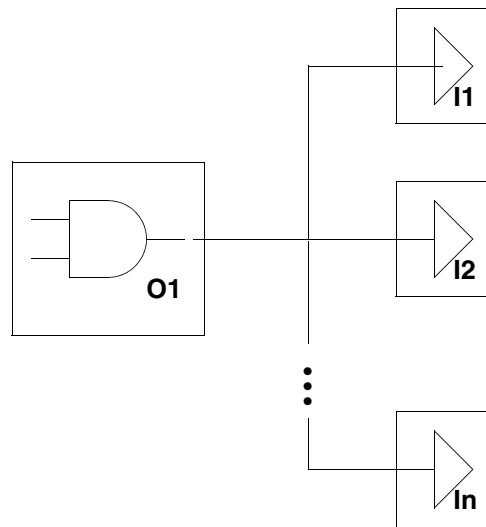
The `balance_buffer` command provides the best results when your library uses linear delay models. If your library uses nonlinear delay models, the second approach provides better results.

- If the large loads reside across the hierarchy from several modules, apply design rules to fix the problem. For example,

```
source constraints.con
compile_ultra
set_max_capacitance 3.0
compile -only_design_rule
```

In rare cases, hierarchical structure might disable Design Compiler from fixing design rules.

In the sample design shown in [Figure 3-3](#), net O1 is overloaded. To reduce the load, group as many of the loads (I1 through In) as possible in one level of hierarchy by using the `group` command or by changing the HDL. Then you can apply one of the approaches.

Figure 3-3 Heavily Loaded Net

Performing High-Effort Compile

The `compile_ultra` command supports two options, which encapsulate higher-effort synthesis strategies. The `-area_high_effort_script` option and `-timing_high_effort_script` option include encapsulated scripts that offer additional area and timing improvements. Depending on the optimization goal, the scripts apply a compile strategy that might turn on or off different optimization features.

The `compile` command supports the `map_effort -high` option. The optimization result depends on the starting point. Occasionally, the starting point generated by the default compile results in a local minimum solution, and Design Compiler quits before generating an optimal design. A high-effort compile might solve this problem.

The high-effort compile uses the `-map_effort high` option of the `compile` command on the initial compile (on the HDL description of the design).

A high-effort compile pushes Design Compiler to the extreme to achieve the design goal. A high-effort compile invokes the critical path resynthesis strategy to restructure and remap the logic on and around the critical path.

This compile strategy is CPU intensive, especially when you do not use the incremental compile option, with the result that the entire design is compiled using a high map effort.

Performing a High-Effort Incremental Compile

You can often improve compile performance of a high-effort compile by using the incremental compile option. Also, if none of the previous strategies results in a design that meets your optimization goals, a high-effort incremental compile might produce the desired result.

An incremental compile (`-incremental_mapping` compile option) allows you to incrementally improve your design by experimenting with different approaches. An incremental compile performs only gate-level optimization and does not perform logic-level optimization. The resulting design's performance is the same or better than the original design's.

This technique can still require large amounts of CPU time, but it is the most successful method for reducing the worst negative slack to zero. To reduce runtime, you can place a `dont_touch` attribute on all blocks that already meet timing constraints.

This incremental approach works best for a logic library that has many variations of each logic cell.

Disabling Total Negative Slack Optimization

You can choose to enable area optimization at the risk of worsening timing violations on some paths and creating some new timing violations, as long as the violation on the most critical path in each path group is not affected. Assuming you are willing to take this risk, you can direct Design Compiler to enable area optimizations on all paths, using the `set_max_area` command with its `-ignore_tns` option. (TNS means total negative slack, the sum of the delay violations of all violating endpoints.) The command line is

```
set_max_area 0 -ignore_tns
```

Optimizing Across Hierarchical Boundaries

The `compile_ultra` command performs the following types of boundary optimization by default:

- Propagation of constants across the hierarchy
- Propagation of equal and opposite information across the hierarchy
- Propagation of unconnected port information across the hierarchy
- Pushing of inverters across the hierarchy (phase inversion)

For information about controlling optimization across hierarchical boundaries, see [“Optimizing Across Hierarchical Boundaries” in Chapter 4](#).

Isolating Input and Output Ports

The `set_isolate_ports` command inserts isolation logic at specified input or output ports. You isolate input and output ports to improve the accuracy of timing models.

Input ports are isolated in the following cases:

- When they drive one or more input pins of a cell having several input pins (as a result of boundary optimization)
- When they drive one or more input pins belonging to different cells having several input pins

Output ports are isolated in the following cases:

- To ensure that the cell driving an output port does not also drive some internal logic within the design
- To specify particular driver cells at the output ports

This is useful when you want each output port to be driven explicitly by its own driver (no sharing of output drivers by two or more ports) or when you want to compile the design in the context of the environment in which the design will be used.

The syntax is

```
set_isolate_ports object_list
    [-type buffer | inverter]
    [-driver
cell_name]
    [-force]
```

object_list

The list of input or output ports you want to isolate.

`-type buffer | inverter`

Specifies whether Design Compiler should use a buffer or a pair of inverters as the isolation logic.

`-driver cell_name`

Specifies a particular cell from the target library as the isolation cell.

-force

With input ports, `-force` indicates that isolation logic is inserted after the specified input port.

With output ports, `-force` indicates that isolation logic is inserted even if no internal feedback from the output drivers occurs.

The isolation logic can be a buffer or a pair of inverters. Either Design Compiler selects the buffer or inverter from the target library, or you specify a particular cell from the target library.

Note:

However, a user-specified cell must be a buffer or an inverter. Otherwise, Design Compiler outputs an error message.

The inserted isolation logic has the `size_only` attribute assigned to it. When you compile the design, therefore, only sizing optimization is allowed for this logic.

Note:

If the isolation logic is composed of an inverter pair, the `size_only` attribute is assigned only to the second inverter, which allows flexibility in optimizing the first inverter.

For example, to insert the isolation logic cell IVDAP on all output ports of the current design, enter the following command:

```
dc_shell> set_isolate_ports [all_outputs] -driver IVDAP
```

You issue the `set_isolate_ports` command before the `compile` command. That is, the isolation logic is inserted before the design is compiled. [Example 3-1](#) shows a sample script.

Example 3-1

```
set target_library lsi_10k.db
set link_library {* lsi_10k.db}
read_verilog ./t1.v
current_design test1
link
set_isolate_ports [all_outputs] -driver IVDAP
compile -map_effort medium
```

It is important to understand that port isolation can be applied only to the input or output ports of the *current* design. Therefore, to apply port isolation to a subdesign of your top-level design, you must first make the subdesign the current design.

Port isolation is currently intended for use only during bottom-up compilation. That is, isolating hierarchical instance pins of lower-level designs from the top-level design in a top-down compilation is not supported.

Also, in a bottom-up compilation, you cannot simply isolate the ports of a subblock that you have temporarily designated as the current design and then expect that isolation logic automatically to propagate upward when you compile the top-level design. To ensure that

the isolation logic of a subblock remains while the top-level design is compiled, you must use the `propagate_constraints` command to propagate the constraints upward after the subblock ports are isolated and before you compile the top-level design.

The following script fragments shows you how to compile a subdesign with port isolation, followed by a top-level compile.

```
current_design test1
link
set_isolate_ports [all_outputs] -type buffer -force
compile -map_effort medium
current_design top
propagate_constraints
compile -map_effort medium
```

Port isolation does not work if

- A `dont_touch` net is connected to the port
- The specified isolation cell is not in the target library
- The specified port is not an output or input port (inout ports and tristate ports are not supported)
- The specified type option (in the `set_isolate_ports` command) is not a buffer or an inverter

You can remove the port isolation attribute from designs by using the `remove_isolate_ports` command. The isolation cells are then removed during the next compile. You can also use the `remove_attribute` command.

To obtain a list of isolated input or output ports in a design, use the `report_isolate_ports` command. The `report_constraint` and `report_compile_options` command also provide information about the isolated input or output ports.

When you issue the `report_isolate_ports` command, you see a report similar to the following:

```
*****
Report : isolate_ports
Design : top
Version: Y-2006.06
Date   : Mon May 1 17:05:43 2006
*****
```

Port Name	Cell Name	Inst. Name	Type	Forced Insertion
stp	IVDA	U35	buffer	yes
rhcp	IVP	U34	inverter	yes
hip_1	IVDA	U32	buffer	no

Other commands that support the port isolation feature are

- `reset_design`

This command removes the port isolation attribute, as well as other attributes from the design.

- `write_script`

This command writes any `set_isolate_ports` commands (along with the other `dc_shell` commands) into the script file.

For more information on the commands discussed in this section, see the appropriate man pages.

Propagating Constants

Design Compiler automatically propagates constants, unconnected pins, and complement information when you use the `compile_ultra` command unless you disable boundary optimization. To learn how to enable constant propagation when boundary optimization is disabled and how to disable constant propagation through specific hierarchical pins, see [“Controlling Constant Propagation” in Chapter 4](#).

Enabling Critical Path Resynthesis

The `compile` command `-map_effort high` option implements an optimization strategy called critical path resynthesis. Critical path resynthesis seeks to resolve timing violations on critical paths by resynthesizing the logic on the paths. The goal is to create a small partition of cells on and around part of the current critical path, then restructure and remap the new partition. The new partition is then accepted or rejected, based on the Design Compiler cost vector.

(The critical path is the path in the design that limits the clock speed—the path with the minimum amount of slack. You can list the cells on the critical path by using the `report_timing` command.)

The `-map_effort high` option is CPU-intensive.

Logic Duplication and Mapping to Wide-Fanin Gates

The DC Ultra version of Design Compiler includes algorithms that enable mapping to wide-fanin gates, plus extensive logic duplication steps on the critical path. Use the `compile_ultra` command or the `-map_effort high` option of the `compile` command to invoke these algorithms.

During compile, Design Compiler evaluates the cells along critical paths. The tool determines whether it can improve the timing or area of the paths by replacing groups of cells with complex, wide-fanin cells from the logic library. Additionally, Design Compiler tries to improve the timing of high-fanout or heavily loaded nets, by duplicating and restructuring large sections of the logic driving the nets. This duplication and restructuring to gain timing improvements often results in significant increases in the design area.

Removing Hierarchy

Design Compiler provides several commands for removing design hierarchy:

`set_ungroup`

Ungroups one or more designs, subdesigns (cells), or references during compilation.

`ungroup`

Ungroups a design or reference manually.

`compile -ungroup_all`

Ungroups all subdesigns during optimization (described earlier in this chapter).

`compile -auto_ungroup area`

Automatically ungroups small hierarchies in the current design and its subdesigns. Use the `compile_auto_ungroup_area_num_cells` variable to specify the minimum number of child cells that a design hierarchy must have so that it is not ungrouped.

`compile -auto_ungroup delay`

Automatically ungroups hierarchies along the critical path in the current design and its subdesigns. Use the `compile_auto_ungroup_delay_num_cells` variable to specify the minimum number of child cells that a design hierarchy must have so that it is not ungrouped.

`compile_ultra`

Automatically ungroups logical hierarchies. Ungrouping merges subdesigns of a given level of the hierarchy into the parent cell or design. It removes hierarchical boundaries

and allows Design Compiler to improve timing by reducing the levels of logic and to improve area by sharing logic.

Optimizing for Multiple Clocks Per Register

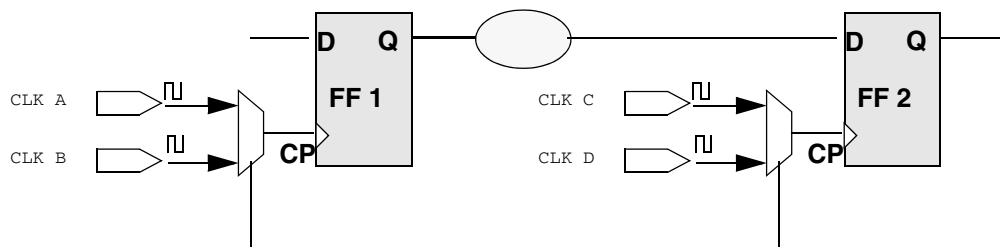
When a sequential device is driven by different clocks, you can either restrict the optimization to one clock at a time or allow multiple clocks to propagate to the sequential device.

To restrict the propagation of clocks so that exactly one reaches the register, you can use either of the following commands: `set_case_analysis` or `set_disable_timing`. You use these commands when you want to select a single active clock for timing analysis and optimization; this method requires multiple iterations for all clocks.

To allow multiple clocks to propagate to a register, set the `timing_enable_multiple_clocks_per_reg` variable to true. Setting this variable to true eliminates the need for multiple iterations to optimize each clock. By default, optimization is restricted to a single clock.

For example, consider the circuit in [Figure 3-4](#). In previous versions of Design Compiler, either CLK A or CLK B could be active; similarly, either CLK C or CLK D could be active. With the multiple clocks per register feature, you can allow Design Compiler to optimize for the following: CLK A to CLK C, CLK A to CLK D, CLK B to CLK C, and CLK B to CLK D.

Figure 3-4 Multiple Clocks per Register



To allow multiple clocks to propagate to a register:

1. Set the `timing_enable_multiple_clocks_per_reg` variable to true. The default is false.

If you do not set this variable to true and more than one clock signal reaches a register, Design Compiler uses one of the clocks propagating to the register.

2. To define multiple clocks at a pin or port, use the `-add` option with the `create_clock` command.

This option allows you to add clocks to the existing clocks; otherwise the existing clocks are overwritten.

If your design has generated clocks, use the following options with the `create_generated_clock` command:

- The `-add` option to specify multiple generated clocks on the same source or pin. Ideally, one generated clock must be specified for each clock that fans into the master pin.
- The `-master_clock` option to specify which clock is the master clock.
- The `-source` option to specify the clock source from which the clock is generated, that is, the pins or ports where the clock waveform is applied to the design.

Additionally, when you use the `-add` option along with the `create_clock` and `create_generated_clock` commands, you must use the `-name` option so that clocks that are defined on the same pin or port have unique names.

3. (Optional) Define a network latency for each clock signal that passes through a specific pin or port to a set of fanout registers.

When multiple clock signals traverse the same pins of a design, any register clock pins that are in the fanout of these clock signals are assigned the same network latency.

You can, however, set clock network latency on a pin or port with reference to a specific clock. To do so, use the `-clock` option with the `set_clock_latency` command. When computing the network latency along the path from a clock definition point to a register clock pin, Design Compiler uses the network latency value associated with the pin or port closest to the register, ignoring network latencies that do not reference the clock of interest.

Important:

For latch-based designs that have more than two clocks, it is strongly recommended that you set false paths between mutually exclusive clocks; otherwise you might observe longer runtime and higher memory usage. Unrelated clocks are those that do not interact with one another. For example, consider [Figure 3-4](#). By default, Design Compiler analyzes the interactions between all combinations of clocks. However, the logic enables only two possible interactions: CLK A to CLK C and CLK B to CLK D. Therefore, set false paths between the unrelated clocks as follows:

```
set_false_path -from CLK A -through FF1/CP -to CLK D
set_false_path -from CLK B -through FF1/CP -to CLK C
```

4. To report registers with multiple clock pins, use the `check_timing -multiple_clock` command and `report_clock` command.

The `check_timing` command generates a warning message if the `timing_enable_multiple_clocks_per_reg` variable is set to false and more than one clock signal reaches a register. The `report_clock` command provides you with the following information about a generated clock: the name of the master clock, the name of the master clock source pin, and the name of the generated clock pin.

Example

The following commands create two clocks on the same port and associate a network latency with each clock signal traversing a particular pin. The commands also set false paths between unrelated clocks.

```
set timing_enable_multiple_clocks_per_reg TRUE
create_clock -name CLKA -period 10 [get_ports CLK]
create_clock -name CLKB -period 8 -add [get_ports CLK]
set_clock_latency 1.16 -clock CLKA [get_pins b1/Z]
set_clock_latency 1.26 -clock CLKB [get_pins b1/Z]
set_input_delay .85 -clock CLKA [get_ports d]
set_input_delay .95 -clock CLKB -add_delay [get_ports d]
set_false_path -from CLKA -to CLKB
set_false_path -from CLKB -to CLKA
```

Preserving the Clock Network After Clock Tree Synthesis

To preserve a clock network after clock tree synthesis, use the `set_dont_touch_network` command. This command sets a `dont_touch_network` attribute on a net group. When placed on a clock tree, the `dont_touch_network` attribute ensures that your clock network is preserved during subsequent optimizations.

To list the clock networks in the design, use the `report_transitive_fanout -clock_tree` command.

Starting at the specified source object, the `set_dont_touch_network` command propagates the `dont_touch_network` attribute throughout the hierarchy of the clock network. By default, the propagation stops at output ports, or at sequential components if setup and hold relationships exist. If you use the `-no_propagate` option, the propagation stops at any logical cell.

The propagation of the `dont_touch_network` attribute occurs only in a forward direction, starting from the specified source object and spreading to objects driven by the source. The propagation can not go backward, even to electrically connected nets in the same net group. This method of propagation highlights an important difference between the `set_dont_touch_network` and the `set_ideal_network` commands: the `set_ideal_network` command propagates in both the forward and backward directions.

For example, if your design is represented by the simple circuit in [Figure 3-5](#), and you issue the following command:

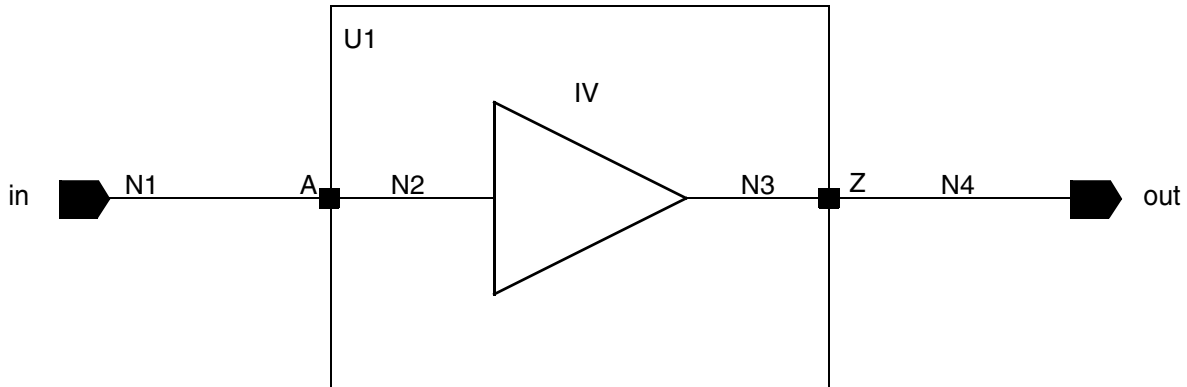
```
set_dont_touch_network U1/A
```

the `dont_touch_network` attribute is propagated from U1/A to U1/N2, U1/IV, U1/N3, and N4. If you issue the command

```
set_dont_touch_network U1/A -no_propagate
```

the `dont_touch_network` parameter is propagated from U1/A to U1/N2. It is not propagated past net U1/N2.

Figure 3-5 `set_dont_touch_network -no_propagate` option



Note:

Use the `set_auto_disable_drc_nets` command when you do not have an accurate representation of the clock tree and you want Design Compiler to treat the clock net as ideal. Typically, you would use this command when a clock tree has not yet been inserted in your design.

The `set_dont_touch_network` command cannot be used if the network has unmapped logic.

You can use the `get_attribute` command to check if an object has either the `dont_touch_network` or the `dont_touch_network_no_propagate` attribute.

For more information on attributes, see the *Design Compiler User Guide*.

Optimizing Once for Best- and Worst-Case Conditions

Using Design Compiler, you can constrain your design *once* for both minimum (best-case) and maximum (worst-case) optimization and timing analysis. The tool then optimizes and analyzes the timing in a single compile run.

You can constrain the design by using either a single logic library or multiple libraries. Whether you use one or multiple libraries, the general methodology is as follows:

1. Set up a logic library file or files. Make sure the files contain

- Best- and worst-case operating conditions
- Optimistic and pessimistic wire load models
- Minimum and maximum timing delays

If you use multiple libraries, see the next section, [“Optimizing With Multiple Libraries” on page 3-20](#).

2. Specify minimum and maximum constraints.

- Environmental—including operating conditions and wire loads
- Clock information—including clock skew and clock transition
- Optimization—including input and output delays, drive, load, and resistance
- Design rule—including transition time, fanout, and capacitance

3. Optimize the design (run `compile`) for simultaneous minimum and maximum timing. To ensure that minimum delay constraints are optimized with respect to a particular clock, specify the `fix_hold` attribute for that clock, using the `set_fix_hold` command.

4. Report and analyze the paths showing constraint violations.

The following constraint-related, reporting, and back-annotation commands support both minimum and maximum optimization and timing analysis.

Constraint-Related Commands

```
set_min_library
set_operating_conditions
set_wire_load_model
set_wire_load_mode
set_wire_load_min_block_size
set_wire_load_selection_group
set_clock_uncertainty
set_clock_transition
```

```
set_drive
set_load
set_port_fanout_number
set_resistance
```

The `set_min_library` command is described in the next section, “[Optimizing With Multiple Libraries](#).”

Reporting Commands

```
report_annotated_delay
report_area
report_attribute
report_bus
report_cache
report_cell
report_clock
report_clusters
report_compile_options
report_constraint
report_delay_calculation
report_design
report_design_lib
report_fsm
report_hierarchy
report_internal_loads
report_lib
report_name_rules
report_net
report_path_group
report_port
report_power
report_qor
report_reference
report_resources
report_synlib
report_timing
report_timing_requirements
report_transitive_fanin
report_transitive_fanout
report_wire_load
```

Other commands, including `report_design`, `report_port`, and `report_wire_load`, generate reports on the minimum and maximum constraints on your design. You do not need to specify the `-min` option.

Optimizing With Multiple Libraries

The `set_min_library` command directs Design Compiler to use multiple technology libraries for minimum- and maximum-delay analyses in one optimization run. Thus, you can choose libraries that contain all of the following:

- Best- and worst-case operating conditions
- Optimistic and pessimistic wire load models
- Minimum and maximum timing delays

You can direct Design Compiler to analyze them simultaneously. To accomplish this analysis, use the `set_min_library` command to create a link between the data in the two libraries. Use the library file name (not the library name) with the `set_min_library` command.

The `set_min_library` command creates a minimum/maximum relationship between two library files. You specify a *max_library* to be used for maximum delay analysis and a *min_library* to be used for minimum delay analysis. Only *max_library* should be used for linking and as the target library.

When Design Compiler needs to compute a minimum delay value, it first analyzes the library cell in the *max_library*, then it looks to the *min_library* to determine if a match exists. If a library cell with the same name, the same pins, and the same timing arcs exists in the *min_library*, Design Compiler uses the timing information from the *min_library*. If the tool cannot find a matching cell in the *min_library*, it uses the cell in the *max_library*.

The syntax is

```
set_min_library max_library
               -min_version min_library | -none
```

Note:

For more information, see the `set_min_library` man page.

Example

[Example 3-2](#) shows how you might use `set_min_library` with `set_operating_conditions` to control and report delay analysis. If you do not specify a minimum, Design Compiler uses the maximum condition for both minimum and maximum delay analysis. You cannot use the `-min` option without also using the `-max` option.

Example 3-2 Controlling and Reporting Delay Analysis

```
set link_library "LIB_WC_COM.db"
set target_library $link_library
set_min_library LIB_WC_COM.db -min_version LIB_BC_COM.db
set_operating_conditions -max WC_COM -min BC_COM
```



```

source minmax.cons
set_fix_hold clk
compile
report_timing -delay_type max
report_timing -delay_type min

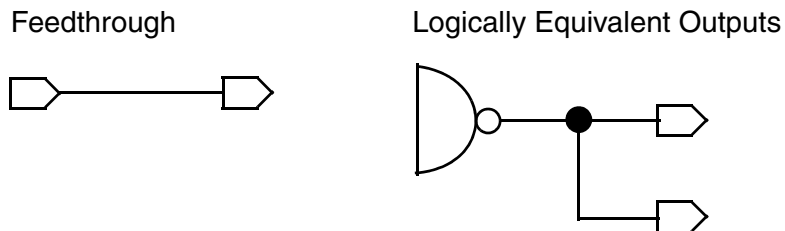
```

Fixing Nets Connected to Multiple Ports

By default, Design Compiler does not buffer nets that are connected to multiple ports. As shown in [Figure 3-6](#), multiple-port connections are nets that connect:

- An input port to an output port (feedthrough) or
- Multiple output ports (logically equivalent outputs)

Figure 3-6 Multiple-Port Connection Types



To represent such nets, Design Compiler uses `assign` statements in the gate-level netlist. Back-end tools could potentially have problems with `assign` statements in the netlist.

To prevent multiple-port connections, use the `set_fix_multiple_port_nets` command to set the `fix_multiple_port_nets` attribute on a design.

You must use the `set_fix_multiple_port_nets` command before compiling. Design Compiler then inserts extra logic (buffers or inverters) that prevent feedthrough nets and multiple-output ports from connecting to the same net. Use options to this command as follows:

- `-feedthroughs` to insert buffers so that input ports are isolated from output ports. A feedthrough net occurs when an input port and output port are connected directly with no intervening logic.
- `-outputs` to insert buffers so that no cell driver pin drives more than one output port.
- `-constants` to duplicate constant logic so that no constant drives more than one output port.
- `-buffer_constants` to buffer logic constants instead of duplicating them.

Design Compiler also has an advanced multiple-port net and constant-driven port fixing capability. Using this capability, Design Compiler first rewires the connections elsewhere in the hierarchy, and any remaining feedthroughs, nets with logically equivalent outputs, and constants that could not be fixed by rewiring are fixed by buffering.

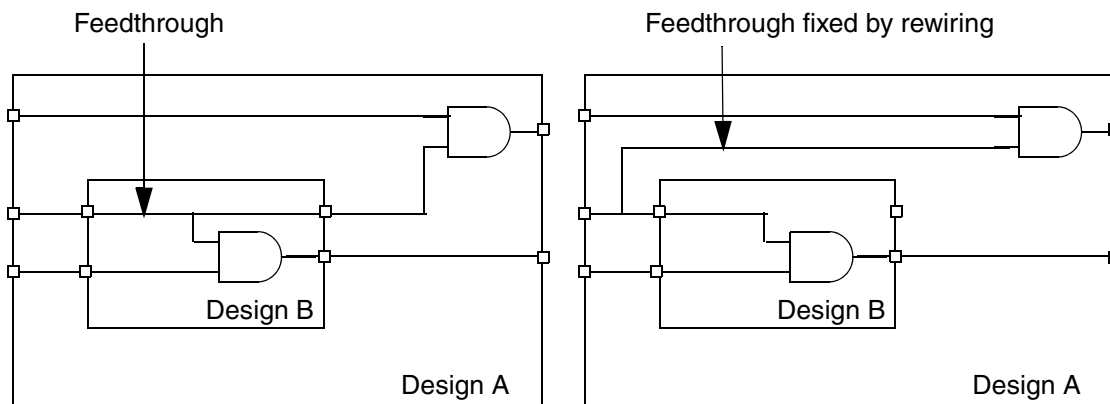
To enable this feature, set the `compile_advanced_fix_multiple_port_nets` variable to `true`. (It is set to `false` by default.) The `compile_rewire_multiple_port_nets` variable must also be set to `true`. (It is set to `true` by default.) When both variables are set to `true`, Design Compiler rewires the connections up through the hierarchy.

To disable the rewiring capability and use buffering to fix the multiple-port nets and constant-driven ports, set the `compile_rewire_multiple_port_nets` variable to `false`.

Design Compiler does not rewire across the hierarchy if boundary optimization is disabled.

The following figure shows a feedthrough in Design B that is fixed by rewiring up through the hierarchy. The feedthrough wire is moved up one level of hierarchy and reconnected to the net in Design A:

Figure 3-7 Feedthrough Fixed By Rewiring Up Through the Hierarchy



For more information, see the man pages.

Building a Balanced Buffer Tree

The `balance_buffer` command builds balanced buffer trees on user-specified nets and drivers of a mapped design. You use this command to fix design rule violations and improve timing delays caused by high fanout nets.

The `balance_buffer` command first removes any existing buffer tree on the specified net or driver and then builds a new buffer tree, free of design rule violations. The tree can span hierarchies downstream of the specified net or driver. However, the `balance_buffer` command does not change the hierarchical pin configuration to balance out loads across the

hierarchies. Within each hierarchy, Design Compiler builds the buffer tree, taking into account the loads and drivers of the next downstream hierarchy.

The generated buffer tree consists of layered stages of buffers or inverters. Each stage uses the same buffer or inverter, and each gate of a given stage drives roughly the same load capacitance. The buffer and inverter cells are optimally chosen by Design Compiler from the logic library, unless you use the `-prefer` option to specify a particular cell from the logic library.

Technology libraries provide various kinds of delay models, including linear and nonlinear models. The `balance_buffer` command can use any delay model provided by the logic library to build the buffer tree.

When all loads have approximately the same value, balanced buffering creates an optimal buffer tree. When loads are not equal, balanced buffering still creates an optimal buffer tree with respect to the delay and design rule constraints, but the tree structure might not be balanced.

An input port driving a net to be buffered by `balance_buffer` must have its drive value set. If you omit the value, no buffer tree is created, because the default value implies that the input port has infinite drive.

Balanced buffering is constraint driven. It fixes design rules first, then optimizes for timing and area.

Note:

The `balance_buffer` command is not recommended for clock trees because the command does not take clock skew into account.

You can use the `-force` option to force buffer tree construction. But if you do, it is possible for the `balance_buffer` command to build buffer trees that worsen design cost. Note that if the design has no design rules or timing cost, buffer trees are not built even with the `-force` option specified.

To perform a functional verification or comparison between the initial mapped design and the mapped design after the `balance_buffer` command is applied, you can use the Formality tool. For more information, see [Chapter 13, “Verifying Functional Equivalence.”](#)

You can display the balanced buffer tree and its level information at a given driver pin by using the `report_buffer_tree` command, and you can remove a buffer tree by using the `clean_buffer_tree` command. For more information about these commands, see their respective man pages.

Example

In this example, a buffer tree is first built from port `io`, and then two additional buffer trees are built to drive `load1` and `load2`.

```
dc_shell> set_driving_cell -lib_cell IV io
```

```
dc_shell> balance_buffer -from io
dc_shell> balance_buffer -to {load1 load2}
```

Defining a Signal for Unattached Master Clocks

Design Compiler can connect master clock pins to a specified signal when it translates or optimizes to flip-flops that have master- and slave-clock pins. In your HDL code, you describe the master-slave latch as a flip-flop by specifying only the slave clock. Specify the master clock as an input port but do not connect it. In addition, set the `clocked_on_also` attribute on the master clock port. Design Compiler then maps the logic to a master-slave cell in the library.

You use the `set_attribute` command to set the `signal_type` attribute to `clocked_on_also` on the master clock port. Design Compiler then maps the logic to a master-slave cell in the library. For multiple clock designs, you use the `-associated_clock` option to specify the associated slave clock.

Example 1

This example illustrates how you use the `set_attribute` command to describe a master-slave latch with a single master-slave clock pair.

```
module MSDFF (Q, D, MCLK, SCLK)

input D, MCLK, SCLK;
output Q;

reg Q;

//synopsys_dc_tcl_script_begin
//set_attribute -type string MCLK signal_type
//                                clocked_on_also
//set_attribute -type boolean MCLK level_sensitive true
//synopsys_dc_tcl_script_end

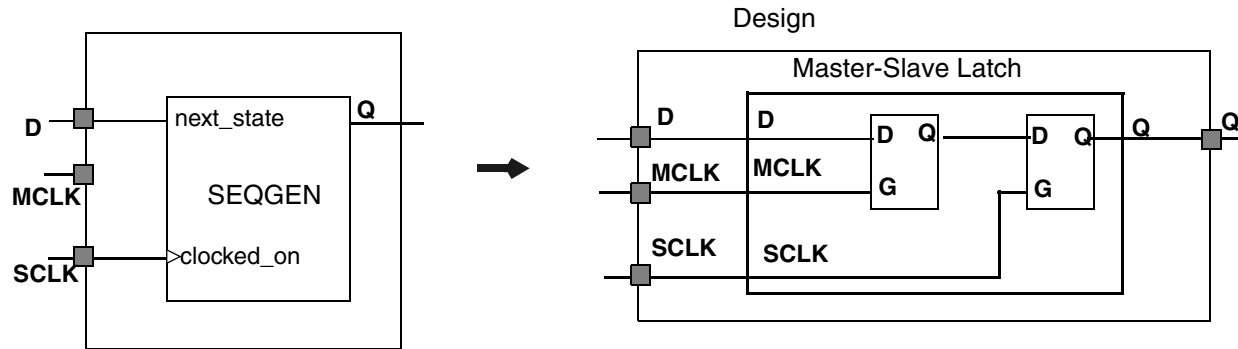
always @ (posedge SCLK)
    Q <= DATA;
endmodule
```

Alternatively, instead of embedding the `set_attribute` command in the RTL, you can include the following commands in your script:

```
create_clock -period 10 [get_ports SCLK]
set_attribute -type string MCLK signal_type clocked_on_also
set_attribute -type boolean MCLK level_sensitive true
```

Figure 3-8 shows the generic cell inferred by Design Compiler and the resultant master-slave latch after compile. As shown in the figure, unattached master clocks are connected to the MCLK port.

Figure 3-8 Generic Cell and Master-Slave Latch After Compilation



Example 2

This example illustrates how you use the `-associated_clock` option to specify the associated slave clock in multiple clock designs. Slave clock port SCK1 and master clock port MCK1 are paired; SCK2 and MCK2 are also paired. The `compile_ultra` or `compile` command automatically connects unconnected master clock pins of cells in the fanout of SCK1 to port MCK1.

```
dc_shell> set_attribute -type string MCK1 signal_type clocked_on_also
dc_shell> set_attribute -type boolean MCK1 level_sensitive true
dc_shell> set_attribute -type boolean MCK1 associated_clock SCK1
dc_shell> set_attribute -type string MCK2 signal_type clocked_on_also
dc_shell> set_attribute -type boolean MCK2 level_sensitive true
dc_shell> set_attribute -type boolean MCK2 associated_clock SCK2
```

See the HDL Compiler documentation for more information on describing master-slave latches.

4

Optimizing Across Hierarchical Boundaries

Boundary optimization and automatic ungrouping are the strategies by which Design Compiler optimizes logic across subdesigns when the design has levels of hierarchy. Boundary optimization performs optimization across design hierarchies while retaining the levels of hierarchy and provides Design Compiler with opportunities to simplify the logic. Ungrouping removes levels of hierarchy and allows Design Compiler more freedom to share common terms in the design.

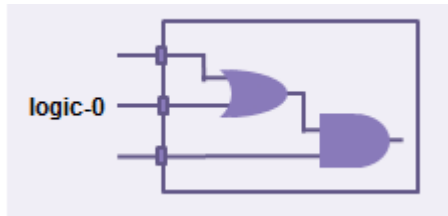
To learn how to control these optimization techniques, see

- [Boundary Optimization](#)
- [Port Punching in Design Compiler Graphical](#)
- [Port Punching and Phase Inversion With Automatic High-Fanout Synthesis](#)
- [Other Optimizations That Affect Hierarchical Boundaries](#)
- [Automatic Ungrouping](#)

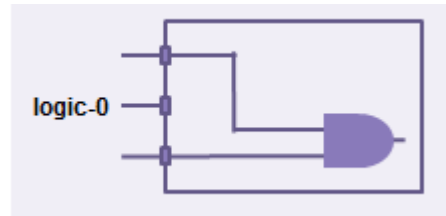
Boundary Optimization

The `compile_ultra` command performs the following types of boundary optimization by default:

- Propagation of constants across the hierarchy:

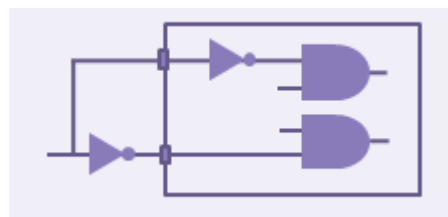


Without boundary optimization

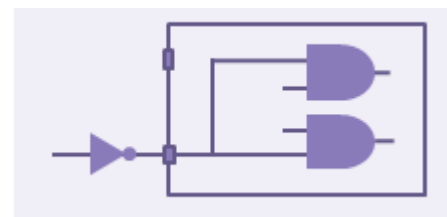


With boundary optimization

- Propagation of equal and opposite information across the hierarchy:

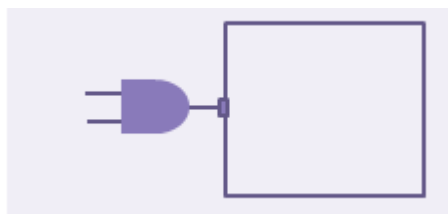


Without boundary optimization

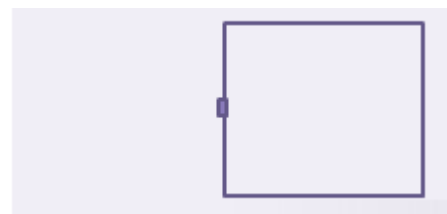


With boundary optimization

- Propagation of unconnected port information across the hierarchy:

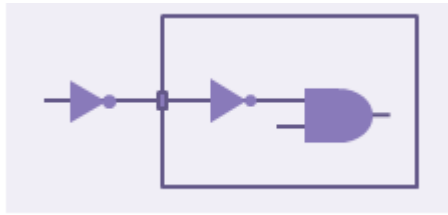


Without boundary optimization

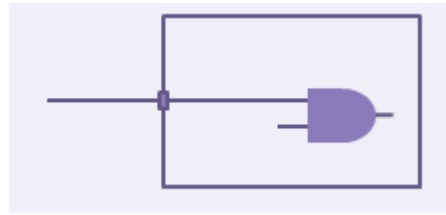


With boundary optimization

- Pushing of inverters across the hierarchy (phase inversion):



Without boundary optimization



With boundary optimization

To learn how to control boundary optimization, see

- [Disabling Boundary Optimization Throughout the Design](#)
- [Disabling Boundary Optimization for a Specific Design](#)
- [Controlling Constant Propagation](#)
- [Controlling Phase Inversion](#)

Disabling Boundary Optimization Throughout the Design

To disable all types of boundary optimization for the entire design, use the `compile_ultra` command with the `-no_boundary_optimization` option.

You do not need to use the `compile_preserve_subdesign_interfaces` variable to disable the downward and upward propagation of constant and equal and opposite information when you use the `-no_boundary_optimization` option with the `compile_ultra` command.

In addition, when you use the `compile_ultra` command with the `-no_boundary_optimization` option, Design Compiler automatically sets the `-no_autoungroup` option, which disables automatic ungrouping for the entire design.

Disabling Boundary Optimization for a Specific Design

To disable all types of boundary optimization on a list of objects (cells or designs), use the `set_boundary_optimization` command. When you run the command, specify the `false` argument for a specific cell or design to disable boundary optimization for that cell or design.

The following example shows how to use the `set_boundary_optimization` command to preserve hierarchical boundaries for a cell named U2:

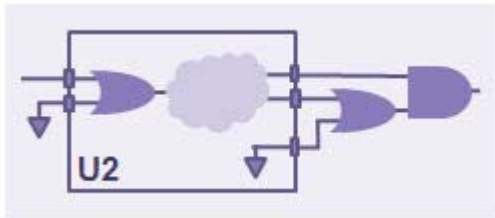
```
dc_shell-topo> set_boundary_optimization [get_cells U2] false
```

When boundary optimization is disabled by the `set_boundary_optimization` command, only the downward propagation of constant and equal and opposite information is disabled. The information is not propagated from a design to its subdesign; however, the information is propagated from a subdesign to its parent design by default.

You can disable both downward and upward propagation of constant and equal and opposite information across the hierarchy by setting the `compile_preserve_subdesign_interfaces` variable to `true`. The default for this variable is `false`.

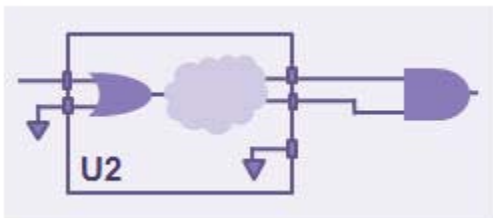
Figure 4-1 shows a design before boundary optimization. Figure 4-2 shows the same design after boundary optimization is performed by the `compile_ultra` command. In this example, the `boundary_optimization` attribute is set to `false` on cell U2 by the `set_boundary_optimization` command.

Figure 4-1 Design Before Boundary Optimization

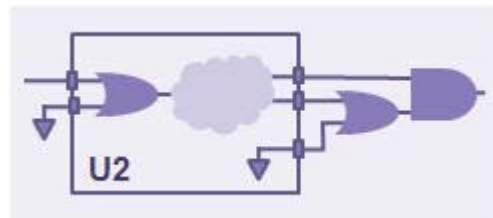


The design on the left in Figure 4-2 shows the default behavior, which disables only the downward propagation of constant and equal and opposite information. In the design on the right, both downward and upward propagation of constant and equal and opposite information is disabled by the `compile_preserve_subdesign_interfaces` variable set to `true`.

Figure 4-2 Design After Boundary Optimization With the `compile_ultra` Command



`compile_preserve_subdesign_interfaces`
set to false (default)



`compile_preserve_subdesign_interfaces`
set to true

When you use the `set_boundary_optimization` command and specify the `false` argument, Design Compiler automatically prevents designs and cells from being ungrouped when you run the `compile_ultra` command.

Controlling Constant Propagation

Design Compiler automatically propagates constants, unconnected pins, and complement information when you use the `compile_ultra` command unless you disable boundary optimization. To learn how to enable constant propagation when boundary optimization is disabled and how to disable constant propagation through specific hierarchical pins, see

- [Enabling Constant Propagation When Boundary Optimization is Disabled](#)
- [Disabling Constant Propagation Through Specific Hierarchical Pins](#)

Enabling Constant Propagation When Boundary Optimization is Disabled

You can enable constant propagation even when boundary optimization is disabled. To propagate constants across hierarchical boundaries even when boundary optimization is disabled, set the `compile_enable_constant_propagation_with_no_boundary_opt` variable to `true`. When you set the variable to `true`,

- The `compile_ultra -no_boundary_optimization` command performs constant propagation throughout the design.
- The `compile_ultra` command without the `-no_boundary_optimization` option performs constant propagation throughout the design, including the blocks that have the `boundary_optimization` attribute set to `false` on them by the `set_boundary_optimization` command.

Disabling Constant Propagation Through Specific Hierarchical Pins

You can prevent the propagation of constants through specific hierarchical pins by using the `set_compile_directives` command with the `-constant_propagation` option set to `false` for a specific hierarchical pin or a list of hierarchical pins. When you do this, Design Compiler sets a `const_prop_off` attribute on the specified pins and disables constant propagation through them.

Controlling Phase Inversion

You can control phase inversion, which is the moving of inverters across hierarchical boundaries during boundary optimization, by using the `compile_disable_hierarchical_inverter_opt` variable.

To prevent inverters from moving across hierarchical boundaries during boundary optimization, set the `compile_disable_hierarchical_inverter_opt` variable to `true`. When the variable is set to `true`, Design Compiler does not move inverters across

hierarchical boundaries even if doing so could improve the design. The variable is set to `false` by default, allowing phase inversion. This variable has no effect on the design when boundary optimization is disabled.

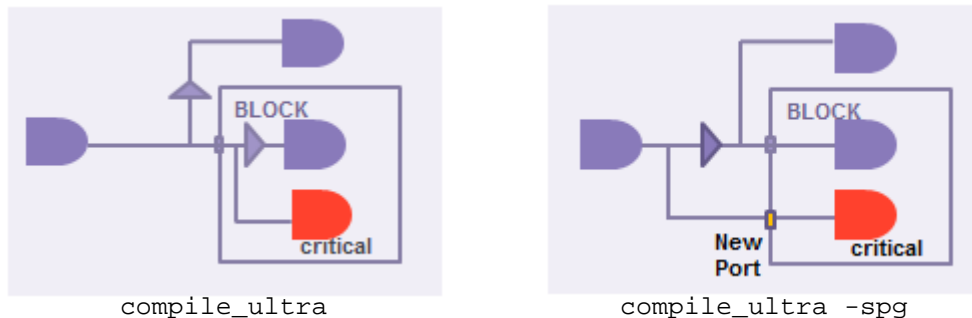
During phase inversion, Design Compiler adds a `_BAR` suffix to the port name. The `port_complement_naming_style` variable defines the convention for renaming ports complemented as a result of boundary optimization. However, It is not recommended that you use the `port_complement_naming_style` variable because it is not supported in Formality.

Port Punching in Design Compiler Graphical

Design Compiler Graphical performs port punching for more flexible load balancing across the hierarchy and improved QoR when you use the `compile_ultra -spg` command.

Figure 4-3 shows buffering across the hierarchy with the `compile_ultra` command (the design on the left) compared to buffering across the hierarchy with the `compile_ultra -spg` command (the design on the right).

Figure 4-3



To disable port punching on the entire design, use the `compile_ultra` command with the `-no_boundary_optimization` option. To disable port punching on specific instances, use the `set_boundary_optimization` command, and set it to `false` for the specific instances.

Port punching is disabled also when you disable port punching with automatic high-fanout synthesis by using the `set_ahfs_options` command with the `-enable_port_punching false` option.

Port Punching and Phase Inversion With Automatic High-Fanout Synthesis

Automatic high-fanout synthesis performs port punching and phase inversion in Design Compiler topographical mode. By default, automatic high-fanout synthesis inserts buffers or inverters across hierarchy boundaries to achieve better QoR.

Port punching in automatic high-fanout synthesis is disabled under the following conditions:

- When you disable boundary optimization by using the `compile_ultra` command with the `-no_boundary_optimization` option, Design Compiler automatically disables port punching and phase inversion for the entire design.
- When you use the `set_boundary_optimization` command and set it to `false` for specific instances, port punching and phase inversion is disabled for those instances.
- When you use the `set_ahfs_options` command with the `-enable_port_punching` option set to `false`, port punching is disabled during automatic high-fanout synthesis.

When you use the `set_ahfs_options` command with the `-preserve_boundary_phase` option set to `true`, inverters cannot move across hierarchical boundaries and the logical phase of the boundary is unchanged during automatic high-fanout synthesis.

Other Optimizations That Affect Hierarchical Boundaries

Design Compiler performs boundary optimization for nets connected to multiple ports and constant-driven ports when the advanced multiple-port net and constant-driven port fixing capability is enabled. For more information, see [“Fixing Nets Connected to Multiple Ports” on page 3-21](#).

Hierarchical clock gating also performs optimization across hierarchical boundaries. For information, see the *Power Compiler User Guide*.

Automatic Ungrouping

The `compile_ultra` command automatically ungroups logical hierarchies. Ungrouping merges subdesigns of a given level of the hierarchy into the parent cell or design. It removes hierarchical boundaries and allows Design Compiler to improve timing by reducing the levels of logic and to improve area by sharing logic.

You can also manually ungroup hierarchies by using the `ungroup` command or the `set_ungroup` command followed by the `compile_ultra` command. For information about manually ungrouping hierarchies, see “Removing Levels of Hierarchy” in the *Design Compiler User Guide*.

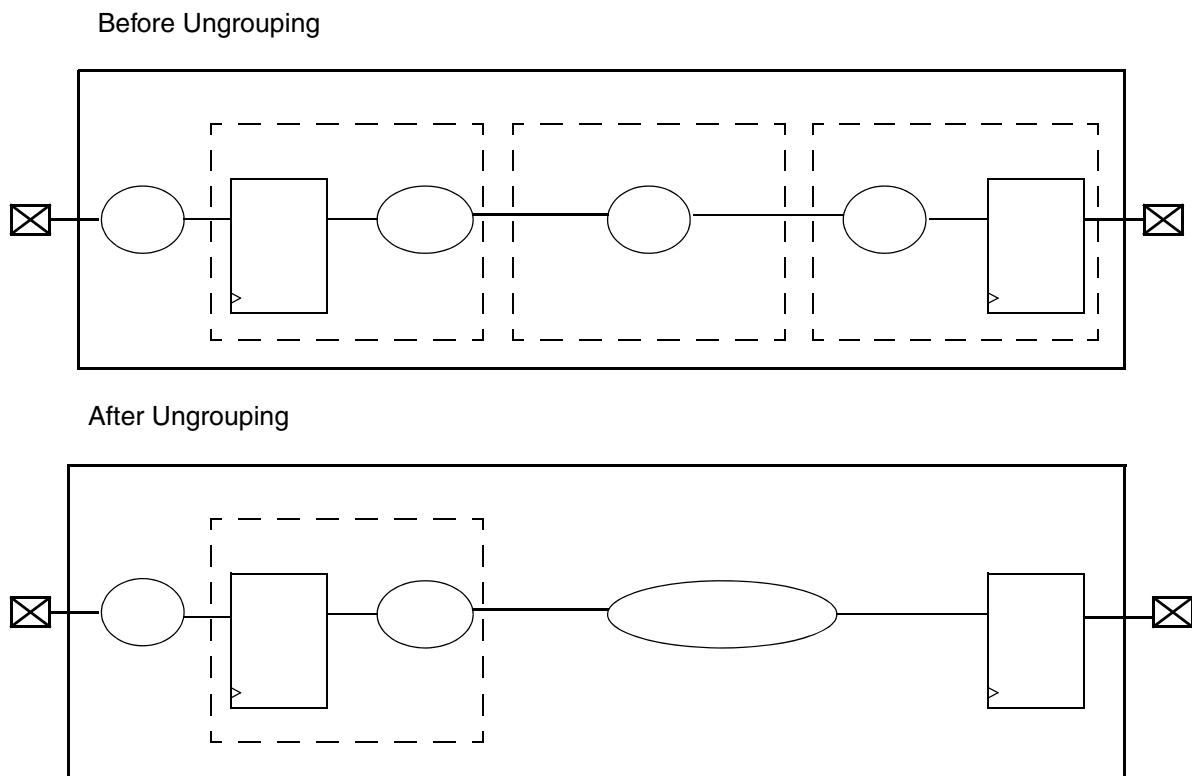
To learn more about automatic ungrouping, see

- [Automatic Ungrouping of Hierarchies](#)
- [Automatic Ungrouping of Designs With Timing Exceptions](#)
- [Exceptions to Automatic Ungrouping](#)

Automatic Ungrouping of Hierarchies

Figure 4-4 shows the hierarchy before and after ungrouping.

Figure 4-4 Automatic Ungrouping of Hierarchies



During optimization, Design Compiler performs the following types of automatic grouping:

- Area-based automatic ungrouping

Before initial mapping, the `compile_ultra` command performs area-based automatic ungrouping. The tool estimates the area for unmapped hierarchies and removes small subdesigns; the goal is to improve area and timing quality of results. Because the tool performs automatic ungrouping at an early stage, it has a better optimization context.

Additionally, datapath extraction is enabled across ungrouped hierarchies. These factors improve the timing and area quality of results.

- Delay-based automatic ungrouping

During delay optimization, the `compile_ultra` command performs delay-based automatic ungrouping. It ungroups hierarchies along the critical path and is used essentially for timing optimization.

- QoR-based automatic ungrouping

When you use the `compile_ultra` command with the `-spg` option, Design Compiler Graphical ungroups additional hierarchies to improve QoR.

To disable automatic ungrouping on a hierarchical design and its subdesigns, use the `-no_autoungroup` option with the `compile_ultra` command. To disable automatic ungrouping on specific instances, use the `set_ungroup` command to set the `ungroup` attribute to `false` on specific designs, cells, or references.

Automatic Ungrouping of Designs With Timing Exceptions

When preserving timing constraints, Design Compiler reassigns the timing constraints to appropriate adjacent, persistent pins (that is, pins on the same net that remain after ungrouping). The constraints are moved forward or backward to other pins on the same net. Note that the constraints can be moved backward only if the pin driving the given hierarchical pin drives no other pin. Otherwise the constraints must be moved forward.

If the constraints are moved to a leaf cell, that cell is assigned a `size_only` attribute to preserve the constraints during a compile. Thus, the number of `size_only` cells can increase, which might limit the scope of the optimization process. To counter this effect, when both the forward and backward directions are possible, Design Compiler chooses the direction that helps limit the number of newly assigned `size_only` attributes to leaf cells.

You can disable this behavior by setting the `auto_ungroup_preserve_constraints` variable to `false`. The default is `true`.

Hierarchies are ungrouped when the following timing constraints are set on hierarchical pins:

- `set_false_path`
- `set_multicycle_path`
- `set_min_delay`
- `set_max_delay`
- `set_input_delay`

- `set_output_delay`
- `set_disable_timing`
- `set_rtl_load`
- `create_clock -period`

Exceptions to Automatic Ungrouping

Hierarchies are not automatically ungrouped in the following cases:

- The hierarchy has user-specified constraints such as `dont_touch`, `size_only`, or `set_ungroup` attributes.
- When boundary optimization is disabled, automatic ungrouping is also disabled.
- In non-topographical mode, the wire load model for the hierarchy is different from the wire load model of the parent hierarchy.

You can override this behavior by setting the `compile_auto_ungroup_override_wlm` variable to `true`. The default is `false`. When the variable is set to `true`, the ungrouped child cells of the hierarchy inherit the wire load model of the parent hierarchy.

Consequently, the child cells might have a more pessimistic wire load model.

5

High-Level Optimization and Datapath Optimization

High-level optimizations are performed when you use the `compile` command or the `compile_ultra` command. The `compile_ultra` command explores additional optimization opportunities; it also performs automatic datapath extraction and advanced datapath transformations.

During high-level optimization, Design Compiler performs arithmetic simplifications and resource sharing. A resource is an arithmetic or comparison operator read in as part of an HDL design. A datapath block contains one or more resources that are grouped and optimized by a datapath generator. During the high-level optimization phases, resources are allocated and shared, depending on timing and area considerations. Resource sharing enables the tool to build one hardware component for multiple operations, which typically reduces the hardware required to implement your design.

Before you read this chapter, read the [“Optimization Flow” on page 1-9](#) to understand how high-level optimizations and datapath optimizations fit into the overall compile flow.

This chapter contains the following sections:

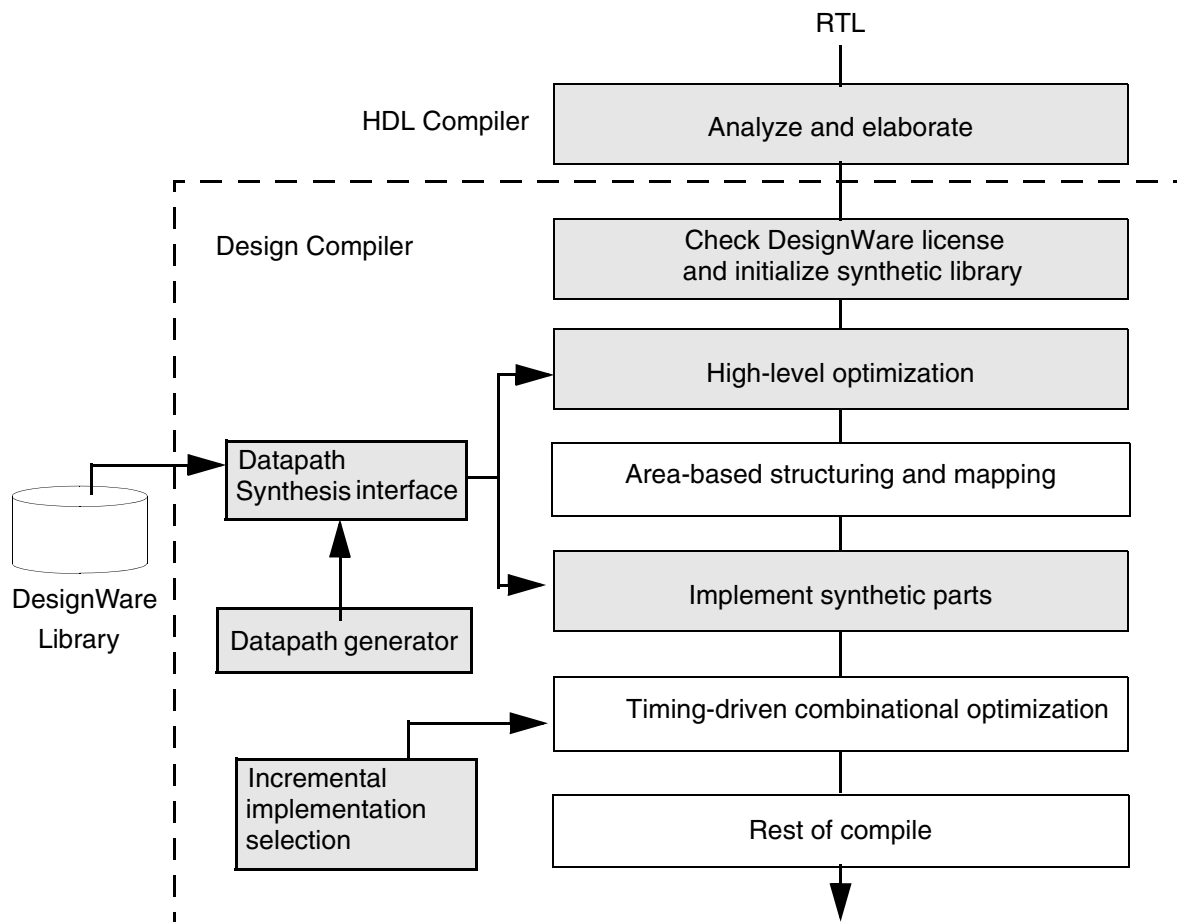
- [Design Compiler Arithmetic Optimization](#)
- [Synthetic Operators](#)
- [Checking DesignWare Licenses](#)
- [High-Level Optimizations](#)

- [Datapath Optimization With DC Ultra](#)
- [Reporting Resources and Datapath Blocks](#)
- [Analyzing Datapath Extraction](#)

Design Compiler Arithmetic Optimization

Figure 5-1 shows how Design Compiler optimizes arithmetic components within the optimization flow described in “[Optimization Flow](#)” on page 1-9. The shaded boxes pertain to the arithmetic optimization flow. Information on these steps is presented in both “[Optimization Flow](#)” on page 1-9 and the sections that follow in this chapter.

Figure 5-1 Optimization of Arithmetic Expressions



The steps in the Design Compiler arithmetic optimization flow are as follows:

1. When HDL Compiler elaborates a design, it maps HDL operators (either built-in operators like + and * or HDL functions and procedures) to synthetic (DesignWare) operators that appear in the generic netlist. See “[Synthetic Operators](#)” on page 5-4.
2. Design Compiler checks for any required licenses and initializes the synthetic library. See “[Checking DesignWare Licenses](#)” on page 5-5.

3. During high-level optimization, Design Compiler manipulates the synthetic operators and applies optimizations such as arithmetic simplifications and resource sharing. See [“High-Level Optimizations” on page 5-6](#). If you are using DC Ultra, Design Compiler performs automatic datapath extraction. See [“Datapath Optimization With DC Ultra” on page 5-9](#).
4. During the implement synthetic parts phase, the tool maps synthetic modules to architectural representations (implementations). For more information on synthetic parts, see the DesignWare documentation.

Design Compiler uses the datapath generator to implement arithmetic components and generate the best implementations. In addition, if you are using DC Ultra, Design Compiler performs advanced datapath transformations on the extracted datapath blocks. See [“Datapath Optimization With DC Ultra” on page 5-9](#).
5. During incremental implementation selection, Design Compiler explores alternative implementations for each arithmetic component. The tool evaluates and replaces synthetic implementations along the critical path to improve delay cost.

Synthetic Operators

Synopsys provides a collection of intellectual property (IP), referred to as the DesignWare Building Block IP Library, to support the synthesis products. Building Block IP provides basic implementations of common arithmetic functions that can be referenced by HDL operators in your RTL source code.

The DesignWare library is built on a hierarchy of abstractions. HDL operators (either built-in operators like + and *, or HDL functions and procedures) are associated with synthetic operators, which are bound in turn to synthetic modules. Each synthetic module can have multiple architectural realizations, called implementations. For example, when you use the HDL addition operator in a design description, HDL Compiler infers the need for an adder resource and puts an abstract representation of the addition operation into your circuit netlist. See [Figure 5-2 on page 5-5](#).

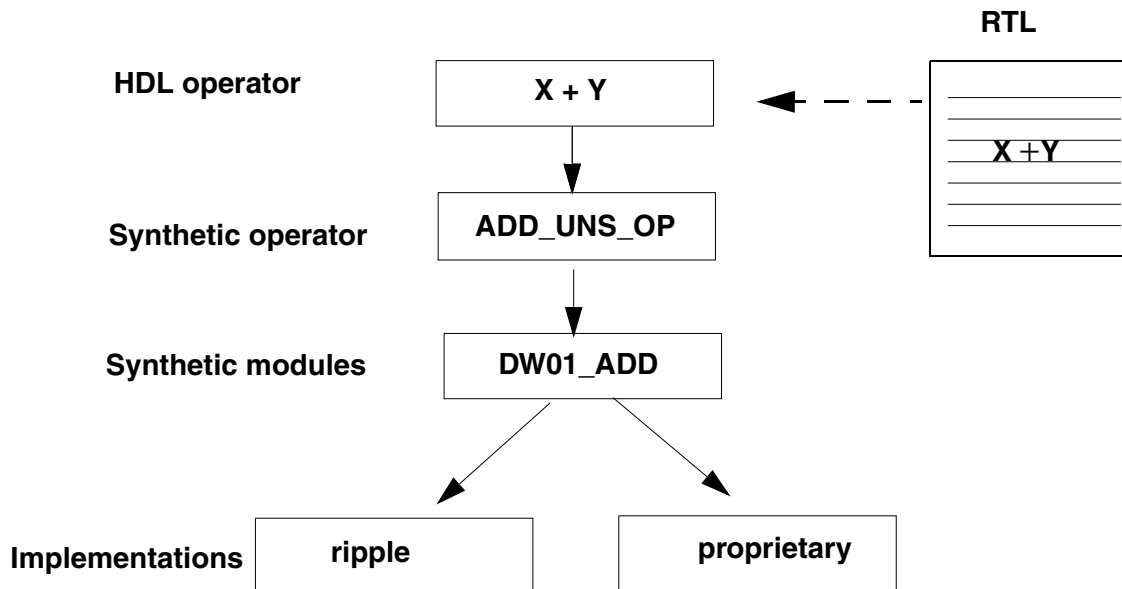
During high-level optimization, Design Compiler manipulates these synthetic operators and applies optimizations such as arithmetic transformations and resource sharing.

To display information about the standard synthetic library that is included with a Design Compiler license, use the `report_synlib` command:

```
report_synlib standard.sldb
```

For more information about DesignWare synthetic operators, modules, and libraries, see the DesignWare documentation.

Figure 5-2 DesignWare Overview



Checking DesignWare Licenses

If any DesignWare component set in the `synthetic_library` variable requires a DesignWare license, Design Compiler checks for this license. You do not need to specify the standard synthetic library, `standard.sldb`, that implements the built-in HDL operators. Design Compiler automatically uses this library.

If you are using additional DesignWare libraries, you must specify these libraries by using the `synthetic_library` variable (for optimization purposes) and the `link_library` variable (for cell resolution purposes) as shown:

```
set synthetic_library {dw_foundation.sldb}
set link_library "*" $target_library $synthetic_library
```

You can force Design Compiler to wait for a DesignWare license by setting the `synlib_wait_for_design_license` variable to DesignWare as follows:

```
set synlib_wait_for_design_license "DesignWare"
```

High-Level Optimizations

During high-level optimization, Design Compiler applies techniques such as tree delay minimization and arithmetic simplifications; it also performs resource sharing.

Tree Delay Minimization and Arithmetic Simplifications

During tree delay minimization, Design Compiler arranges the inputs to arithmetic trees. For example, the expression $a + b + c + d$ describes three levels of cascaded addition operations. The tool can rearrange this expression to $(a + b) + (c + d)$, which might result in faster logic (only two levels of cascaded operations).

Additional simplification is available with the `compile_ultra` command; some examples are as follows:

- Sink cancellation
The expression $(a + b - a)$ is simplified to b .
- Constant folding
The expression $(a * 3 * 5)$ is transformed to $(a * 15)$

Resource Sharing

Resource sharing reduces the amount of hardware needed to implement operators such as addition (+) in your Verilog or VHDL description. Without this feature, each operation is built with separate hardware. For example, every + operator builds an adder. This repetition of hardware increases the area of a design.

There are two basic types of resource sharing: Common subexpression elimination and sharing mutually exclusive operations. For more information about resource sharing, see the following subsections:

- [Common Subexpression Elimination](#)
- [Sharing Mutually Exclusive Operations](#)
- [Enhanced Resource Sharing](#)

Common Subexpression Elimination

This type shares redundant computations in a design. To understand common subexpression elimination, consider [Example 5-1](#).

Example 5-1 Original RTL

```
X = A > B;
Y = A > B && C;
```

This code contains two comparators and a logical add. In common subexpression elimination, the common subexpression, $A > B$, is grouped and the code is transformed to [Example 5-2](#).

Example 5-2 Expression $A > B$ Shared

```
Temp = A > B;
X = Temp;
Y = Temp && C;
```

This transformation reduces the number of comparators from two to one.

Both HDL Compiler and Design Compiler perform common subexpression elimination. However, HDL Compiler does not share the $+$, $*$, and $-$ operators by default because it might reduce the sharing options available to Design Compiler during compile. Design Compiler shares these operators and all other operators by default during timing-driven optimization.

The following operators are shared by default when Design Compiler does common subexpression elimination:

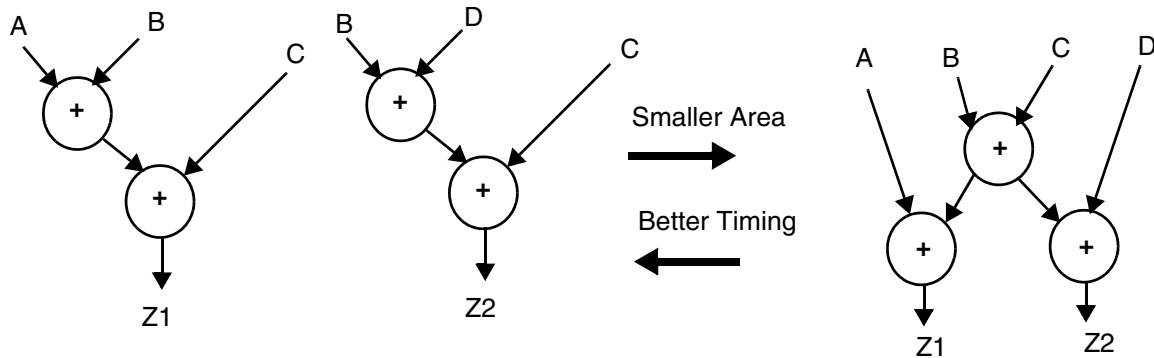
- Relational ($=$, $<$, $>$, $<=$, $>=$, $!=$)
- Shifting ($<<$, $>>$, $<<<$, $>>>$)
- Arithmetic ($+$, $-$, $*$, $/$, $**$, $\%$)

Selectors that drive arithmetic operators

Additionally, if you use the `compile_ultra` command, the tool can identify common subexpressions automatically; you do not need to use parentheses or write them in the same order. For example, the expressions $(A + B + C)$ and $(B + A + D)$, $A + B$ and $B + A$ are recognized as a common subexpression.

Furthermore, the tool can either share common subexpressions or reverse the sharing depending on constraints. Consider the following expressions: $Z1 <= A + B + C$, $Z2 <= B + C + D$, and arrival time is $A < B < D < C$; [Figure 5-3](#) shows how the tool might reverse the sharing of common subexpressions depending on constraints. The tool determines whether to share or reverse the sharing of operators during a later phase—that is, during timing-driven optimization.

Figure 5-3 Sharing and Unsharing of Arithmetic Subexpressions



Sharing Mutually Exclusive Operations

This type of resource sharing shares operators in a single process when there is no execution path that reaches both operators from the start of the block to the end of the block—that is, operations that cannot be performed simultaneously are shared. To understand this type of resource sharing, consider [Example 5-3](#).

Example 5-3 Sharing Two + Operators

```
module resources(A,B,C,SEL);
  input A,B,C,D;
  input SEL;
  output [1:0] Z;

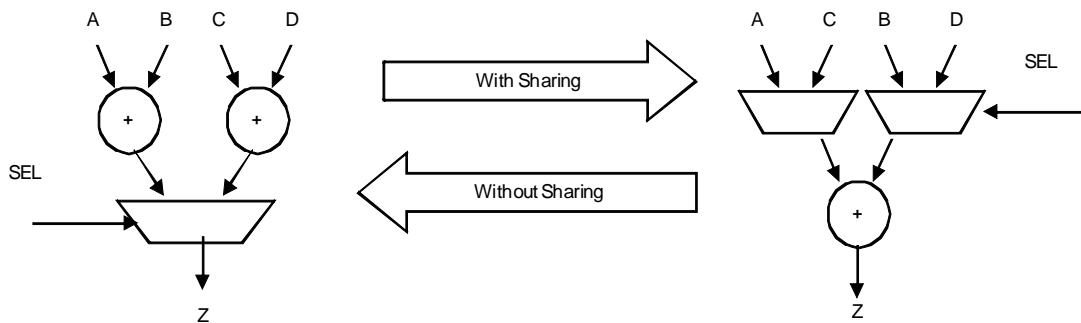
  reg [1:0] Z;

  always @(A or B or C or D or SEL)
  begin
    if(SEL)
      Z = B + A;
    else
      Z = C + D;
  end
endmodule
```

[Example 5-3](#) shows code that adds either $A + B$ or $D + C$; what is added depends on whether the condition `SEL` is true.

Without resource sharing, the tool builds two adders and one MUX, and with resource sharing, the tool uses only one adder to build the design, as shown in [Figure 5-4](#).

Figure 5-4 Design With and Without Sharing



Similar to common subexpression sharing, the tool determines whether to share or reverse sharing depending on timing constraints. For example, when the arrival of the SEL signal is late and sharing the adder worsens timing quality of results (QoR), Design Compiler does not share the adder.

Enhanced Resource Sharing

You can perform enhanced resource sharing by using the `compile_enhanced_resource_sharing` variable. It is useful to enable the variable for designs that contain numerous arithmetic operators. If you set the variable to `true`, Design Compiler identifies more opportunities for sharing in the design, such as common subexpression sharing of nonidentical comparators and sharing of mutually exclusive operators that do not drive the same selector directly. By default, the `compile_enhanced_resource_sharing` variable is set to `false`.

Datapath Optimization With DC Ultra

DC Ultra datapath optimization requires a DC-Ultra license and a DesignWare license. Datapath design is commonly used in applications that contain extensive data manipulation, such as 3-D, multimedia, and digital signal processing (DSP).

DC Ultra datapath optimization is comprised of two steps: Datapath extraction, which transforms arithmetic operators (for example, addition, subtraction, and multiplication) into datapath blocks, and datapath implementation, which uses a datapath generator to generate the best implementations for these extracted components.

During datapath optimization, DC Ultra does the following:

- Shares (or reverses the sharing) of datapath operators
- Uses the carry-save arithmetic technique

- Performs high-level arithmetic optimization on the extracted datapath
- Explores better solutions that might involve a different resource-sharing configuration
- Allows the tool to make better tradeoffs between resource sharing and datapath optimization

Enabling DC Ultra Datapath Optimization

DC Ultra datapath optimization is enabled by default when you use the `compile_ultra` command.

Note:

DC Ultra datapath optimization requires both the DC-Ultra-Features license and the DesignWare-Foundation license. The DesignWare license is pulled when you run the `compile_ultra` command.

Datapath Extraction

Datapath extraction transforms arithmetic operators (for example, addition, subtraction, and multiplication) into datapath blocks to be implemented by a datapath generator. This transformation improves the quality of results (QoR) by utilizing the carry save arithmetic technique.

Carry save arithmetic does not fully propagate carries but instead stores results in an intermediate form. The carry-save adders are faster than the conventional carry-propagate adders because the carry-save adder delay is independent of bit-width. These adders use significantly less area than carry-propagate adders because they do not use full adders for the carry.

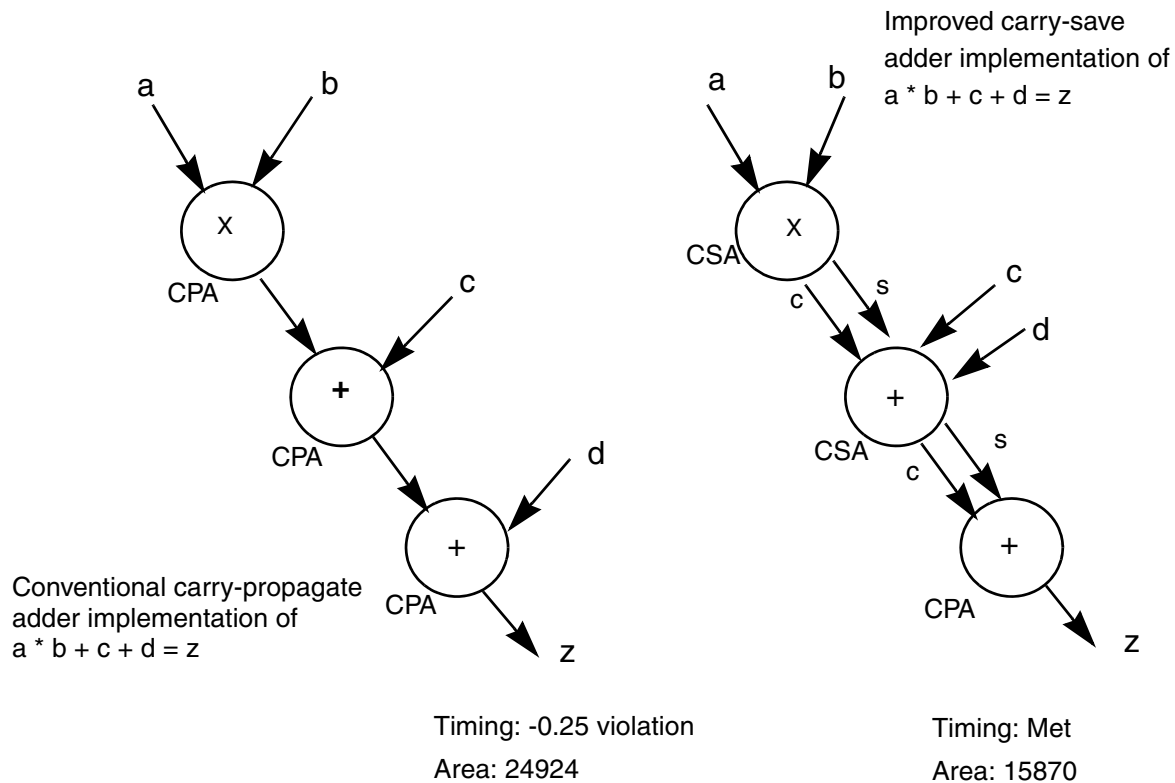
[Example 5-4](#) shows the code for the expression $a * b + c + d = z$. [Figure 5-5](#) shows that the conventional implementation of the expression $a * b + c + d = z$ would use three carry-propagate adders (CPAs); whereas, the carry save technique requires only one carry-propagate adder and two carry-save adders (CSAs). [Figure 5-5](#) also shows the timing and area numbers for both implementations.

Example 5-4

```
module dp (a,b,c,d,e);
  input [15:0] a,b;
  input [31:0] c,d;
  output [31:0] Z;

  assign Z = ( a * b )+ c + d;
endmodule
```

Figure 5-5 Conventional Carry-Propagate Adder and Faster, Smaller Carry-Save Adder



When DC Ultra datapath optimization is used to compile design dp in [Example 5-4](#), the following improvements are realized:

- 8.3 percent timing improvement, compared to DC Expert results
- 36 percent area improvement, compared to DC Expert results

The DC Ultra datapath solution supports extraction of the following components:

- Arithmetic operators that can be merged into one CSA tree
- Operators extracted as part of a datapath: *, +, -, >, <, <=, >=, ==, !=, and MUXes
- Variable shift operators (<<, >>, <<<, >>> for Verilog and sll, srl, sla, sra, rol, ror for VHDL)

- Operations with bit truncation

The datapath flow can extract these components only if they are directly connected to each other—that is, no nonarithmetic logic between components. Keep the following points in mind:

- Extraction of mixed signed and unsigned operators is allowed only for adder trees
- Instantiated DesignWare components cannot be extracted

Datapath Implementation

All the synthetic operators and the extracted datapath elements are implemented by the DesignWare datapath generator in DC Ultra.

The DesignWare datapath generator uses the smart generation technology to perform the following tasks:

- Implement datapath blocks using context-driven optimizations
- Revisit high-level optimization decisions for a given timing context
- Consider logic library characteristics

For further fine tuning, the control of the smart generation strategies is available by using the Design Compiler `set_dp_smartgen_options` command.

For more information on the datapath smart generation strategies, see the man page of the `set_dp_smartgen_options` command and the DesignWare documentation.

Advanced Datapath Transformations with DC Ultra

Design Compiler performs advanced datapath transformations on the extracted datapath blocks. Some examples of advanced datapath transformations performed by DC Ultra as follows:

- Sum-of-products to Product-of-Sums
The expression $(A * C + B * C)$ is transformed to $(A + B) * C$.
- Comparator sharing
Expressions such as $A > B$, $A < B$, $A \leq B$ are transformed to use single subtractors with multiple comparison outputs.
- Optimization of parallel constant multipliers

- Operand reordering
The tool can rearrange operands of multipliers or comparators to produce different quality of results (QoR).
- Explore trade-offs between common subexpression elimination (CSE) sharing and mutually exclusive operations (MUTEX) sharing. Design Compiler can undo the CSE sharing in the GTECH netlist generated by HDL Compiler to facilitate the optimum combination of CSE and MUTEX sharing.

Reporting Resources and Datapath Blocks

Use the `report_resources` command to generate a report that lists the resources and datapath blocks used in the design. To understand the resources report, consider the code in [Example 5-5](#).

Example 5-5 Design add: Code

```
module datapath (a, b, c, d, sel, z1, z2);
input [7:0] a, b, c, d;
input sel;
output [15:0] z1, z2;
wire [15:0] prod = sel? a * b : a * c;
assign z1 = prod + d;
assign z2 = c * d;
endmodule
```

When this code is compiled, the `report_resources` command generates the report shown in [Example 5-6](#).

In this example, the `report_resources` command generates the following three reports:

- Resource report for arithmetic operators that are mapped to individual DesignWare components
- Datapath report for arithmetic operators that are merged into a single datapath block by datapath extraction
- Implementation report for each arithmetic cell

The Resource report shows that a `mult_x_7_0` multiplier cell is mapped to a `DW_mult_unsigned` DesignWare multiplier.

The Datapath report shows that the operators are merged into a single datapath cell, `DP_OP_5_297`. The contained operations show the list of operations that are contained in each cell. Note that the `_5` suffix in the operation names represents the line number in the RTL code in [Example 5-5](#). If more than one operator appears in a line, it is reflected in the suffix. Two operators appear in line 5 in [Example 5-5](#); therefore, the second multiplier is identified in the report with a `_5_2` suffix.

The Datapath report table shows the input (PI) and output (PO) ports of the datapath cell. It also shows the datapath expression with the intermediate fanout (IFO) and its data class. Note that the expression in this report is just a functional representation of each datapath block. The data class shows whether the signal should be zero extended (unsigned) or sign extended (signed) in computation when the signal is used for subsequent operations. The report does not describe the actual internal implementation of the block.

The Implementation report shows the implementation of each DesignWare block. The report includes the implementation name and the optimization mode that is used to implement each DesignWare cell. The implementation report is generated for both detected DesignWare cells and ungrouped DesignWare cells.

Example 5-6 Datapath Report for Design Datapath Generated by the report_resources Command

```
*****
Report : resources
Design : datapath
Version: D-2010.03
Date   : Mon Feb  1 09:19:09 2010
*****
```

Resource Report for this hierarchy in file /usr/.../datapath.v

Cell	Module	Parameters	Contained Operations
mult_x_7_1	DW_mult_uns	a_width=8 b_width=8	mult_7
DP_OP_5_298_491	DP_OP_5_298_491		

Datapath Report for DP_OP_5_298_491

Cell	Contained Operations
DP_OP_5_298_491	mult_5 mult_5_2 add_6

Var	Type	Data Class	Width	Expression
I1	PI	Unsigned	8	
I2	PI	Unsigned	8	
I3	PI	Unsigned	1	
I4	PI	Unsigned	1	
I5	PI	Unsigned	8	
I6	PI	Unsigned	8	
T0	IFO	Unsigned	16	I1 * I2
T1	IFO	Unsigned	16	I1 * I5
T2	IFO	Unsigned	16	{ I3, I4 } ? T0 : T1
O1	PO	Unsigned	16	T2 + I6

Implementation Report

Cell	Module	Current Implementation	Set Implementation
mult_x_7_1	DW_mult_uns	pparch (area,speed)	
DP_OP_5_298_491	DP_OP_5_298_491	str (area,speed)	

Analyzing Datapath Extraction

To get the best QoR results from DC Ultra datapath optimization, it is important that the biggest possible datapath blocks are extracted from the RTL code. See SolvNet article 015771, “Coding Guidelines for Datapath Synthesis” for information about coding styles to help improve datapath extraction:

<https://solvnet.synopsys.com/retrieve/015771.html>

To examine how the datapath blocks are extracted from your RTL before you run the `compile_ultra` command, use the `analyze_datapath_extraction` command. The command analyzes the arithmetic contents of the design and provides feedback for improving the RTL code.

When the `analyze_datapath_extraction` command detects an arithmetic operator that is not extracted, it reports the information about what is blocking the datapath extraction. The following example shows an analysis report about fanout truncation that breaks the datapath block:

```
dc_shell-topo> analyze_datapath_extraction
Information: Checking out the license 'DesignWare'. (SEC-104)
Analyzing datapath extraction ...
Information: Operator associated with resources 'add_8 add_8_2' in design
'test' breaks the datapath block because there is leakage due to
truncation on its fanout. (HDL-120)
```

After the analysis, the report generates lists of the resources and datapath blocks used in the design. The report shows which arithmetic operators in the RTL code are extracted. See “Reporting Resources and Datapath Blocks” on page 5-13 for information about interpreting the resource report, and see Example 5-6 on page 5-14 to see a report example.

6

Multiplexer Mapping and Optimization

Design Compiler can map combinational logic representing multiplexers in the HDL code directly to a single multiplexer (MUX) or a tree of multiplexer cells from the target logic library. Before you read this chapter, read the [“Optimization Flow” on page 1-9](#) to understand how multiplexer mapping and optimization fit into the overall compile flow.

Multiplexers are commonly modeled with if and case statements. To implement this logic, HDL Compiler uses SELECT_OP cells, which Design Compiler maps to combinational logic or multiplexers in the logic library. If you want Design Compiler to preferentially map multiplexing logic to multiplexers—or multiplexer trees—in your logic library, you must infer MUX_OP cells.

The MUX_OP cell should be inferred when you want Design Compiler to build a multiplexer tree structure for the case statement blocks in your HDL. MUXs can be implemented efficiently (in speed and area) in the library. This type of structure can provide advantages in circuit performance and savings in wiring area, compared to implementation constructed from random logic. This feature is supported only with the use of the case statement in VHDL or Verilog code.

During compilation, Design Compiler replaces each MUX_OP cell with the best multiplexer tree implementation of the N-input M-output multiplexer. The implementation depends on the constraints given for the design and the arrival times of the selector inputs.

This chapter contains the following sections:

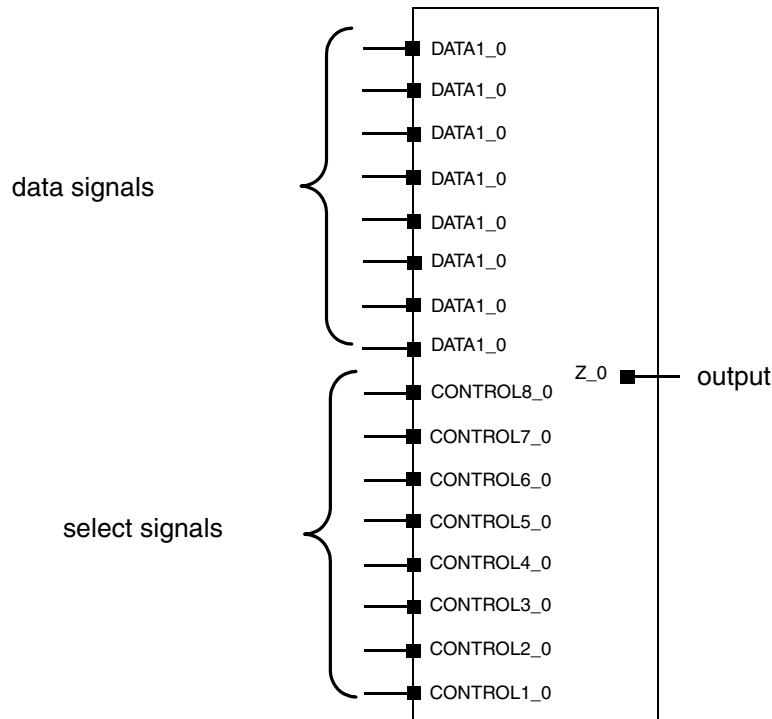
- [Inferring SELECT_OPs](#)
- [Inferring MUX_OPs](#)

- [Library Cell Requirements for Multiplexer Optimization](#)
- [Mapping Multiplexers on Asynchronous Signal Lines](#)
- [Mapping to One-Hot Multiplexers](#)

Inferring SELECT_OPs

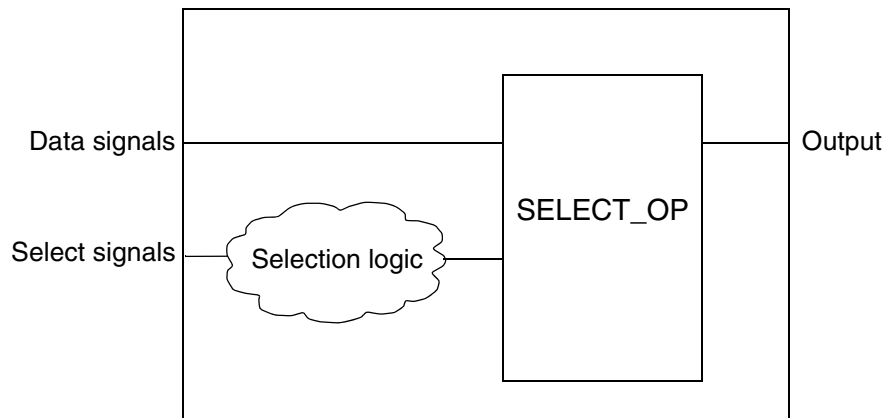
By default, HDL Compiler uses SELECT_OP components to implement conditional operations implied by if and case statements. An example of a SELECT_OP cell implementation for an 8-bit data signal is shown in [Figure 6-1](#).

Figure 6-1 SELECT_OP Implementation for an 8-bit Data Signal



For an 8-bit data signal, 8 selection bits are needed.
This is called a one-hot implementation.

SELECT_OPs behave like one-hot multiplexers; the control lines are mutually exclusive, and each control input allows the data on the corresponding data input to pass to the output of the cell. To determine which data signal is chosen, HDL Compiler generates selection logic, as shown in [Figure 6-2](#).

Figure 6-2 Verilog Output—SELECT_OP and Selection Logic

Depending on the design constraints, Design Compiler implements the SELECT_OP with either combinational logic or multiplexer cells from the logic library. For more information on SELECT_OP inference, see the HDL Compiler documentation.

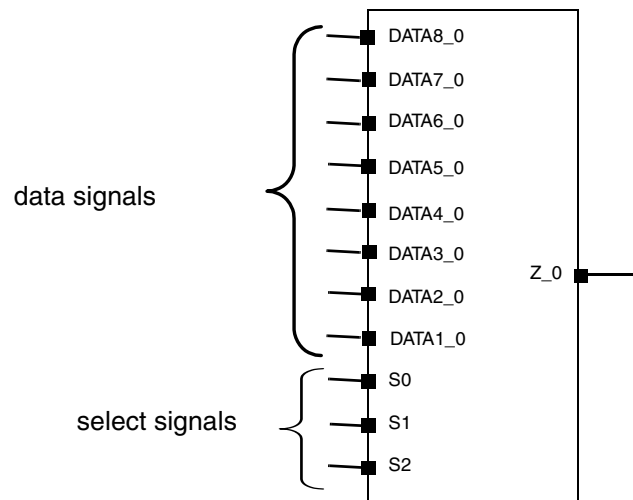
Inferring MUX_OPs

If you want Design Compiler to preferentially map multiplexing logic in your RTL to multiplexers—or multiplexer trees—in your logic library, you need to infer MUX_OP cells. These cells are hierarchical generic cells optimized to use the minimum number of select signals. They are typically faster than the SELECT_OP cell, which uses a one-hot implementation. Although MUX_OP cells improve design speed, they also might increase area. During optimization, Design Compiler preferentially maps MUX_OP cells to multiplexers—or multiplexer trees—from the logic library, unless the area costs are prohibitive, in which case combinational logic is used.

You can embed an attribute or directive in the HDL code or use variables to tell HDL Compiler which part of the HDL description to implement as a single multiplexer or a tree of multiplexers. When the HDL is read in, a generic cell called MUX_OP cell represents the multiplexer functionality. During optimization, Design Compiler maps the logic inside the MUX_OP cell to an implementation, using multiplexer cells from the library.

The MUX_OP cell is a generic representation of an N:1 multiplexer with M output bits. When VHDL or Verilog code is read in, the resulting design contains a MUX_OP cell for every case block inside a process that contains the `infer_mux` directive. A MUX_OP cell also is inferred for each signal (including bused signals) assigned inside the same case block. The signals in the HDL that compute the selector are connected to the select inputs of the MUX_OP cell. [Figure 6-3](#) shows a generic MUX_OP cell for an 8-bit data signal.

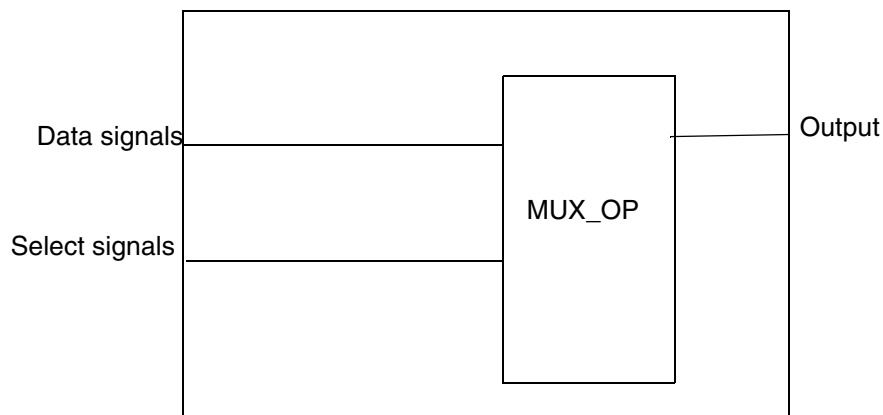
Figure 6-3 MUX_OP Generic Cell for an 8-bit Data Signal



For an 8-bit word, only 3 selection bits are needed.

The MUX_OP cell contains internal selection logic to determine which data signal is chosen; HDL Compiler does not need to generate any selection logic, as shown in [Figure 6-4](#).

Figure 6-4 HDL Compiler Output—MUX_OP Generic Cell for 8-Bit Data



The naming convention for MUX_OP cells in a design is

`_MUX_OP_N_S_M`

Argument	Description
N	The number of data inputs.
S	The number of select inputs.
M	The width of the output signal.

Because the MUX_OP cell is implemented as a level of hierarchy, certain types of logic sharing that might otherwise take place are no longer possible. In some cases, this can lead to a design that is less suitable than one having no MUX_OP cells.

The MUX_OP feature enables you to direct Design Compiler to generate a MUX tree structure for a given case statement when you already know that a MUX tree is the best representation for that statement. Design Compiler optimizes the tree based on constraints and yields the best-possible MUX tree.

To generate MUX_OP cells for a specific case or if statement, use the `infer_mux` directive in the HDL description as shown in [Example 6-1](#). For more information on this directive and inference limitations, see the HDL Compiler documentation.

Example 6-1 *Using the infer_mux directive in the HDL Description*

```
always@(SEL) begin
  case (SEL) // synopsys infer_mux
    2'b00: DOUT <= DIN[0];
    2'b01: DOUT <= DIN[1];
    2'b10: DOUT <= DIN[2];
    2'b11: DOUT <= DIN[3];
  endcase
```

Observe the following when you use MUX_OP cells:

- Do not set the HDL Compiler variable `hdl_infer_mux` to all when you compile. Setting this variable to all results in implementing MUX_OP cells for every case statement in your HDL. MUX_OP cells should be inferred for case statements that can benefit most from a multiplexer tree structure.
- Incompletely specified case statements can result in MUX_OP cells with unused inputs or a partially collapsed multiplexer tree structure. In such cases, the difference between using a MUX_OP cell and implementing the multiplexer with random logic might not be significant.
- In general, most gains are realized when you use a MUX_OP cell for fully specified case statements that implement large multiplexer logic. If your design uses mostly small (2:1 or 4:1) multiplexers, you might get better results by not using MUX_OP cells.

- To ensure that MUX_OP cells are mapped to MUX technology cells, you must apply a `size_only` attribute to the cells to prevent logic decomposition during the optimization steps. You can set the `size_only` attribute on each MUX_OP manually or allow HDL Compiler to set it automatically when reading the RTL. You can control the automatic behavior by using the `hdlin_mux_size_only` variable. For more information about the `hdlin_mux_size_only` variable, see the HDL Compiler documentation.

Design Compiler can also recognize and map to one-hot multiplexers. See [“Mapping to One-Hot Multiplexers” on page 6-8](#).

Library Cell Requirements for Multiplexer Optimization

The multiplexer optimization requires the presence of at least a 2:1 multiplexer cell in the logic library. The inputs or outputs of this cell can be inverted. If a 2:1 multiplexer primitive cell does not exist in the library, you see the following warning message:

```
Warning: Target library does not contain any 2-1 multiplexer.
(OPT-853)
```

An implementation of the MUX_OP cell from the target library is created, but it might not be the best implementation possible. All multiplexer cells in the target library can be used to construct the implementation of the MUX_OP cell except

- Enabled multiplexer cells
- Bused output multiplexer
- Multiplexers larger than 32 : 1

For Design Compiler to make the best use of the multiplexer cells available in your logic library, recompile the library or obtain a library compiled with version V3.4a or later from your ASIC vendor.

Mapping Multiplexers on Asynchronous Signal Lines

Design Compiler can map MUX_OP cells and two-input SELECT_OP cells on asynchronous set and reset signals and clock signals to MUX cells, reducing glitches on asynchronous register pins. By default, the `compile` or `compile_ultra` commands preserve multiplexers in the fanin cone of asynchronous register pins. Specifically, if Design Compiler finds multiplexing logic (any MUX_OP or two-input SELECT_OP cell) in the fanin of only asynchronous register pins (clock or asynchronous set and reset pins), the multiplexing logic is mapped to a multiplexer and a `size_only` attribute is applied to the multiplexer. To turn this capability off, set the `compile_enable_async_mux_mapping` variable to `false`.

You do not need to set the `size_only` attribute, either manually or with the `hdlin_mux_size_only` variable. Design Compiler sets the `size_only` attribute automatically.

If the `MUX_OP` or `SELECT_OP` cell is specified as `dont_touch`, Design Compiler does not map the cell to a multiplexer.

Mapping to One-Hot Multiplexers

Design Compiler also supports inference and mapping of one-hot multiplexers as described in the following sections.

Inferring One-Hot Multiplexers

A one-hot multiplexer is a library cell that behaves functionally as an AND/OR gate such as an AO22 or AO222. The difference is that in case of a one-hot MUX, there are as many control inputs as data inputs and the function of the cell ANDs each control input with the corresponding data input. For example, a 4-to-1 one-hot MUX has the following function:

$$Z = (D_0 \ \& \ C_0) \mid (D_1 \ \& \ C_1) \mid (D_2 \ \& \ C_2) \mid (D_3 \ \& \ C_3)$$

One-hot MUXes are generally implemented using passgates, which makes them very fast and allows their speed to be largely independent of the number of data bits being multiplexed. However, this implementation requires that exactly one control input be active at a time. If no control inputs are active, the output remains floating. If more than one control input is active, there could be an internal drive fight.

Design Compiler allows you to control one-hot MUX inference and mapping. Because of the restriction on the control inputs of a one-hot MUX (that is, one control input is active at all times), Design Compiler cannot automatically make use of these gates. The tool cannot verify that the control inputs behave as required. Hence, it does not automatically map to these cells. Instead, it maps only to cells that you specify can be mapped. Also, after the tool maps these cells, they cannot be unmapped. The cells can only be sized.

[Example 6-2](#) and [Example 6-3](#) show the coding styles that are supported.

Example 6-2

```
case (1'b1) //synopsys full_case parallel_case infer_onehot_mux
sel1 : out = in1;
sel2 : out = in2;
sel3 : out = in3;
```

Example 6-3

```
case({sel3, sel2, sel1}) //synopsys full_case parallel_case
infer_onehot_mux
```



```
001: out = in1;
010: out = in2;
100: out = in3;
```

Note:

The `parallel_case` and `full_case` pragmas are required. The `infer_onehot_mux` pragma is supported only in Verilog and System Verilog.

For more information on coding styles and pragmas, see the HDL Compiler documentation.

Library Requirements for One-Hot Multiplexers

Design Compiler can recognize and map to a one-hot MUX cell in the target library only if the one-hot MUX cell meets with all of the following requirements:

- It is a single-output cell.
- Its inputs can be divided into two disjoint sets of the same size as follows:

$$C = \{C_1, C_2, \dots, C_n\} \text{ and } D = \{D_1, D_2, \dots, D_n\}$$

where n is greater than 1 and is the size of the set. Actual names of the inputs can be different from the connotation shown.

- The `contention_condition` attribute must be set on the cell. The value of the attribute is a combinational function, FC , of inputs in set C that defines prohibited combinations of inputs as shown in the following examples (where the size n of the set is 3):

$$FC = C_0' \& C_1' \& C_2' \mid C_0 \& C_1 \mid C_0 \& C_2 \mid C_1 \& C_2$$

or

$$FC = (C_0 \& C_1' \& C_2' \mid C_0' \& C_1 \& C_2' \mid C_0' \& C_1' \& C_2)'$$

- The cell must have a combinational function FO defined on the output with respect to all its inputs. This function FO must logically define, together with the contention condition, a base function F^* that is the sum of n product terms, where the i th term contains all the inputs in C , with C_i high and all others low and exclusively one input in D . Examples of the defined function are as follows (for $n = 3$):

$$F^* = C_0 \& C_1' \& C_2' \& D_0 \mid C_0' \& C_1 \& C_2' \& D_1 \mid C_0' \& C_1' \& C_2 \& D_2'$$

or

$$F^* = C_0 \& C_1' \& C_2' \& D_0' + C_0' \& C_1 \& C_2' \& D_1' + C_0' \& C_1' \& C_2 \& D_2'$$

The function FO itself can take many forms, as long as it satisfies the following condition:

$$FO \& FC' == F^*$$

That is, when FO is restricted by FC', it should be equivalent to F*. The term $FO = F^*$ is acceptable; other examples are as follows (for $n = 3$):

$$FO = (D_0 \& C_0) \mid (D_1 \& C_1) \mid (D_2 \& C_2)$$

or

$$FO = (D_0' \& C_0) \mid (D_1' \& C_1) \mid (D_2' \& C_2)$$

Note that when FO is restricted by FC, inverting all inputs in D is equivalent to inverting the output; however inverting only a subset of D would yield an incompatible function. Although Design Compiler supports any form of FO that satisfies the condition $FO \& FC' == F^*$, it is recommended that you use a simple form (such as those described previously or F^*).

An example of a properly specified cell is as follows. For more information on cell definition, see the Library Compiler documentation.

Example 6-4 One-hot MUX Cell Definition

```
cell(OHMUX2) {
... ..
contention_condition : "(C0 C1 + C0' C1')";
... ..
pin(D0) {
direction : input;
... ..
}
pin(D1) {
direction : input;
... ..
}
pin(C0) {
direction : input;
... ..
}
pin(C1) {
direction : input;
... ..
}
pin(Z) {
direction : output;
function : "(C0 D0 + C1 D1)";
... ..
}
}
```

Optimization of One-Hot Multiplexers

Because a one-hot MUX implementation requires that exactly one control input be active at a time, composition is not supported. Composition of larger cells from smaller ones requires extra logic to ensure that exactly one control input is active at any time, which is inconsistent with the intention of the use of one-hot MUXes; in addition, the implementation of composition also depends on the actual electronic structure of the library cells.

However, if Design Compiler does not find an exact match in the library, it generates a warning message and maps an RTL MUX to a larger MUX—that is, more inputs—from the library and tie the additional inputs to ground. If the RTL one-hot MUX cannot fit into the largest one-hot MUX cell from the target library, the tool does not perform one-hot MUX mapping and issues a warning message. The MUX in this case is mapped the normal way.

7

Multibit Component Optimization

You can use Design Compiler to synthesize certain logic to multibit components in vendor libraries. A multibit component is a group of cells with identical functionality. Two cells can have identical functionality even if they have different bit-widths. Thus, a group of cells, including one 3-bit register and one 5-bit register, is a multibit component (assuming identical functionality). It would still be called a multibit component if it were implemented using eight single-bit cells. Multibit library cells consume less power and area and create a more uniform layout structure than single-bit equivalents can attain.

This chapter focuses on synthesizing flip-flops and latches to multibit library cells. It contains the following sections:

- [Mapping to Multibit Registers](#)
- [Creating Multibit Components](#)
- [Removing Multibit Components](#)
- [Reporting Multibit Components](#)
- [Using Multibit Register Mapping With an Incremental Compile](#)
- [Library Cell Requirements for Multibit Optimization](#)

Mapping to Multibit Registers

Multibit inference allows you to map registers to multibit library cells. Multibit components offer several benefits, such as reduced area and power and a more regular structure for place and route.

To perform multibit register mapping, follow these steps:

1. Create a multibit component.

When you create a multibit component, you select registers to be mapped to multibit registers either in the RTL or after you read the RTL.

Multibit components can be created in the following ways:

- From the RTL (inferring multibit components for bused registers)

Use the `set_hdlin_infer_multibit` command to control multibit inference for signals that have the `infer_multibit` directive in the Verilog description. Multibit components can be created for bused registers only.

For more information, see the HDL Compiler documentation.

- From `dc_shell`

Use the `create_multibit -name mb_name {object list}` command to infer multibit registers. You can create multibit components for non-bused registers also.

For more information, see [“Creating Multibit Components” on page 7-4](#).

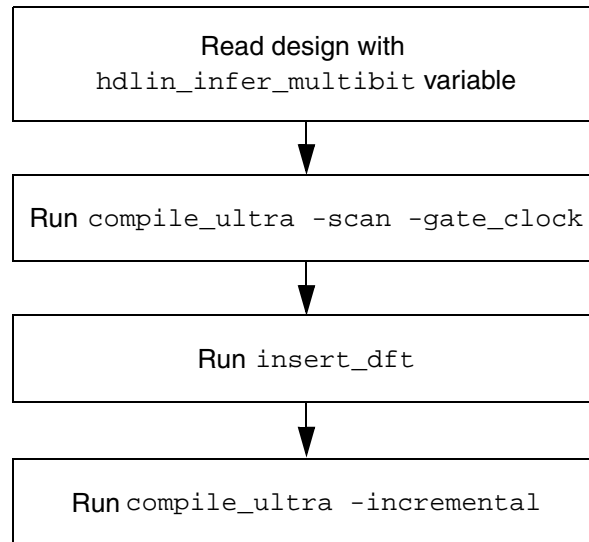
2. Map multibit components to multibit registers.

The multibit component that is created in the previous step is mapped to the multibit registers during compilation if the corresponding multibit registers are available in the target library.

The `set_multibit_options -mode` command allows you to determine how multibit components are optimized during compilation. The default mode for multibit register mapping is `non_timing_driven`, which maps multibit components to multibit registers whenever possible. Use the `timing_driven` mode to map to multibit registers only when there is no impact to timing or area QoR.

Figure 7-1 shows an example of a multibit register mapping flow.

Figure 7-1 Multibit Register Mapping Flow



An example multibit register mapping flow using scan replacement and clock gating is described in the following steps:

1. Set the `hdlin_infer_multibit` variable to `default_all`, and read the design.

2. Apply timing constraints and set the multibit mapping mode. For example, enter

```
prompt> set_multibit_options -mode timing_driven | non_timing_driven
```

The tool uses the `non_timing_driven` mode by default.

3. Run the `compile_ultra` command. For example, enter

```
prompt> compile_ultra -scan -gate_clock
```

Scan replacement and clock gating are not performed for multibit registers during the incremental compile step when you use the `non_timing_driven` and `timing_driven` modes. You need to start the initial `compile_ultra` run with the `-scan` and `-gate_clock` options to enable scan replacement and clock gating.

4. Verify that the multibit registers are used in the design.

Use the `report_multibit` command with the `-hierarchical` option to report the multibit cells in the subdesigns:

```
prompt> report_multibit -hierarchical
```

5. Run the `insert_dft` command to enable scan insertion.

If you want DFT Compiler to treat each bit of the multibit registers individually, you must set the `set_scan_configuration -preserve_multibit_segment` command to `false`. Otherwise, DFT Compiler preserves the entire multibit segment and does not balance the scan chains for each register.

6. Run an incremental compile by using the following command:

```
prompt> compile_ultra -incremental
```

Creating Multibit Components

To create multibit components in the current design after reading the RTL, use the `create_multibit` command. The syntax is `create_multibit object_list` where *object_list* specifies a list of cells in the current design for which a multibit component is to be created.

For example, to create multibit components named `y` for `reg[0]` through `reg[3]` and sort them in 0-2-1-3 order, use the following command:

```
prompt> create_multibit -name y {y_reg[0] y_reg[2] y_reg[1] y_reg[3]} \
      -no_sort
```

You can create multibit components in subdesigns from the top-level design by specifying the instance name of the subdesign, as shown:

```
prompt> create_multibit -name U1_x {U1/x_reg[0] U1/x_reg[1] U1/x_reg[2] \
      U1/x_reg[3]}
```

All registers in the multibit component must be from the same design.

The `create_multibit` command also allows you to create multibit components for registers that are not bused registers:

```
prompt> create_multibit -name MB_reg {a_reg b_reg}
```

For more information, see the `create_multibit` man page.

Removing Multibit Components

To remove multibit components from the current design, use the `remove_multibit` command. The syntax is `remove_multibit object_list` where *object_list* specifies a list of multibit components, cells, and cell instances to remove or detach. The objects must be from the current design. For example, to remove the cell `y_reg[2]` from a multibit component, use the following command:

```
prompt> remove_multibit y_reg[2]
```


To remove multibit component *y* from the current design, use the following command:

```
prompt> remove_multibit y
```

You can remove multibit components in subdesigns from the top-level design by specifying the instance name of the subdesign, as shown:

```
prompt> remove_multibit U1/U1_x
```

For more information, see the `remove_multibit` man page.

Reporting Multibit Components

You can infer a multibit component from the HDL source code by adding directives, or you can create the component from Design Compiler by using the `create_multibit` command. The compilation process preserves multibit components even if their implementations undergo changes.

Use the `report_multibit` command to report all multibit components in your current design. The report lists the multibit component name and the cells that implement each bit. You can use this command on a mapped design or an unmapped design.

Use the `object_list` argument with the `report_multibit` command to specify a list of cells and multibit components to show in the report. If this argument is not specified, all multibit components in the current design are listed. Use the `-hierarchical` option with the `report_multibit` command to report multibit components in all of the subdesigns.

The following excerpt is from a report produced by the `report_multibit` command:

```
Report: Multibit
Design: your_design
...
```

```
Multibit Component : data_reg
Cell                Reference    Library          Area    Width  Attributes
-----
data_reg[0:15]      ff2x16    your_library     48.00    16      n
data_reg[16:31]     ff2x16    your_library     48.00    16      n
-----
Total 2 cells                               96.00    32
-----
```

Design Compiler uses a colon to identify multibit component registers with consecutive bits (0:15 and 16:31 as shown in the report). If the colon conflicts with your back-end tool's naming requirements, change the colon to another delimiter using the `bus_range_separator_style` variable.

The tool uses a comma to separate nonconsecutive bits. For example, if you use bits 0 through 5 and bit 7 in the multibit component, the report lists them as 0 : 5, 7. The `bus_multiple_separator_style` variable controls this delimiter.

To report multibit components in a subdesign, specify the hierarchical instance name of the subdesign. The following example generates a report of multibit components in the instance U1:

```
prompt> report_multibit U1/*
```

The following example generates a report of multibit components in all of the subdesigns:

```
prompt> report_multibit -hierarchical
```

For more information, see the `report_multibit` man page.

Using Multibit Register Mapping With an Incremental Compile

When you review the final QoR after placement, you might want Design Compiler to decompose some multibit registers. To do this,

1. Read the netlist after the initial `compile_ultra` step.

You need to read the netlist before the scan insertion step to apply the `remove_multibit` command. The incremental compile does not reconnect scan chains after decomposing the scan-stitched multibit registers.

2. Set the `remove_multibit` command on the registers you want to decompose to single-bit registers.

3. Run an incremental compile by using the `compile_ultra -incremental` command.

The `remove_multibit` command is the only command that can be used before the incremental compile step. Design Compiler does not create new multibit registers during incremental compilation for multibit components added by the `create_multibit` command.

Library Cell Requirements for Multibit Optimization

Design Compiler supports only multibit library cells that have identical functionality for each bit. The multibit library cell interfaces must be either parallel (bused) or global (shared). For example, if you want to infer a 4-bit banked flip-flop with a shared asynchronous clear signal, the clear signal must be shared among all 4 bits. In this case, Design Compiler can use a multibit register with a bused asynchronous clear signal.

Design Compiler cannot infer a multibit register if the first and second bits share one asynchronous reset signal but the third and fourth bits share another reset signal. In this case, Design Compiler does not infer a multibit flip-flop but uses 4 single-bit flip-flops instead.

Large multibit cells might cause routing congestion or might be too inflexible for your design. To prevent Design Compiler from using multibit library cells larger than 8 bits, use the `set_dont_use` command with the `multibit_width` library attribute. For example,

```
set_dont_use \  
[get_cells -filter "@multibit_width > 8" library_name/*]
```

For more information about multibit library cells, see the *Library Compiler Methodology and Function Modeling User Guide*.

8

Optimizing Finite State Machines

Design Compiler provides unique optimization capabilities for finite state machines (FSMs). Design Compiler extracts and optimizes FSMs automatically. It requires a DC-Ultra license and can be applied to Verilog and VHDL designs, Synopsys state tables (.st files), and already compiled FSM designs. The methodology is referred to as the DC Ultra FSM flow or simply the automatic FSM flow. This flow requires minimal user input. Before you read this chapter, read the [“Optimization Flow” on page 1-9](#) to understand how FSM optimization fits into the overall compile flow.

This chapter contains the following sections:

- [Basic Description of Finite State Machines](#)
- [Synthesizing Finite State Machines](#)
- [Verifying a Finite State Machine](#)
- [Creating Finite State Machine Reports](#)

Basic Description of Finite State Machines

A finite state machine is a circuit for which the primary output values depend not only on the current values of the primary inputs of the circuit but also on the sequence of previous primary input values. An FSM circuit contains flip-flops or registers (banks of flip-flops), as well as combinational cells. It is the presence of both flip-flops and feedback logic that can produce the dependence of the outputs on the sequence of input values, that is, the input sequence (along with the feedback logic) ultimately determines the values held at the output pins of the flip-flops at any given time.

General Behavior of a Finite State Machine

The time-dependent *state vector* of the FSM is defined by a set of flip-flops. The different achievable combinations of bit values that can be held by the flip-flop outputs define the states of the FSM. That is, each state is represented as a unique pattern of logic 0s and 1s stored by the state flip-flops. (Typically, the flip-flops use set and reset signals to produce logic 1 and 0, respectively, at the Q output pin.)

The state vector bit patterns correspond to *state encodings* or state assignments. State encodings affect in important ways both the extraction process and FSM optimization phase. In most cases, Design Compiler can derive these encodings from the input design file. You can also set or reset the encodings by using particular commands. If no state encodings are defined by the input file or user commands, the tool automatically assigns encodings.

The current values held by the state vector flip-flops correspond to a particular state encoding and represent the present state of the FSM. The present state or the present state along with the current values of the primary input produces an associated set of values at the primary outputs. When the primary inputs change and a synchronizing event occurs (usually the rising or falling edge of a clock signal), the state vector changes from the present state to the next state, which can be a new state or the same state. The next state produces its particular set of output values. In this way, as the inputs change, the FSM can sequence through a set of states that generate different outputs.

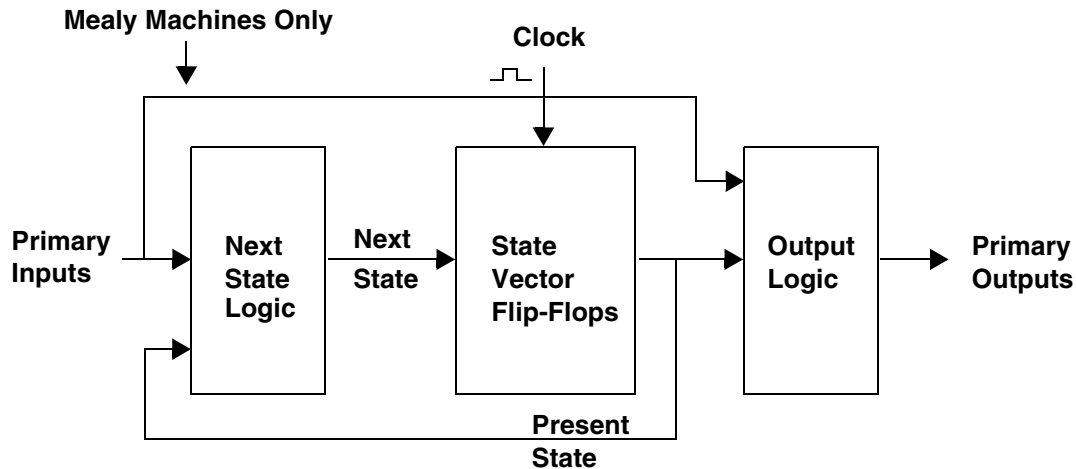
Note that a finite state machine does not necessarily use all possible bit combinations of the flip-flops. Often it does not.

For example, a circuit with N states requires at least $\log_2(N)$ flip-flops. A state vector that is specified with M flip-flops has 2^M possible state encodings. But if the number of valid states, N, in a circuit is less than 2^M (the maximum number of encodable states possible), then some state encodings would not represent valid states. In a correctly defined circuit, however, these unassigned encodings never appear. Instead *don't care conditions* are assigned, which can help Design Compiler simplify the FSM logic during optimization.

Finite State Machine Architecture

Figure 8-1 shows the general structure and behavior of an FSM. As described previously, the architecture for an FSM consists of primary inputs, a set of flip-flops for holding the state vector, and two combinational logic networks: the next state logic and the output logic, leading to the primary outputs.

Figure 8-1 Architecture of a Finite State Machine



There are two basic types of FSMs: the Moore machine, for which the outputs are derived only from the state vector, and the Mealy machine, for which the outputs depend on both the primary inputs and the state vector. Figure 8-1 shows the architecture for both machines.

State Vector, State Encodings, and Encoding Styles

State vectors, state encodings, and state encoding styles are Design Compiler constructs that influence the way the tool optimizes an FSM. Specifically, when the state encodings and the state encoding style are carefully defined with respect to the state vector definition, Design Compiler can take advantage of the don't care and unassigned states to achieve an optimum result during compile.

In the DC Ultra automatic FSM flow, the tool can derive these constructs for most Verilog and VHDL designs, and always for state table designs.

State Vector

The state vector is specified by an *ordered* list of flip-flop instance names. The named flip-flops store an ordered bit pattern that defines the current state of an FSM at any given time. A particular flip-flop bit pattern corresponds to an actual or legal state of the FSM *if* the flip-flop bit pattern maps to the predefined, ordered bit pattern of a particular state encoding

(see the next section on state encodings later). This mapping establishes a one-to-one correspondence between the state assignments (state encodings) and a set of unique flip-flop bit patterns. Any flip-flop bit patterns that do not map to the predefined state encodings are not legal states and are treated as don't care states.

For example, the list {ff0 ff1 ff2}, where ff0, ff1, and ff2 are flip-flop instance names, defines a 3-bit-wide state vector. This state vector can represent an FSM with eight or fewer states, because each flip-flop can hold a 0 or a 1, leading to eight unique bit patterns. A state with the encoding (101) is represented by the state vector bit pattern (101), where ff0 holds the value 1, ff1 holds the value 0, and ff2 holds the value 1. Similarly, the state encoding (001) is represented by the state vector bit pattern (001), where ff0 and ff1 both hold a 0 and ff2 holds a 1. On the other hand, if the state encoding (110) has not been assigned, the state vector (110) is treated as a don't care state.

Note that the length of the state vector and the length of the state encodings must agree.

For most HDL design files, Design Compiler can derive the state vector. For other input design files, where the state vector cannot be derived by the tool, you must define the state vector by using the `set_fsm_state_vector` command.

For example, to specify the 4-bit state vector, where the flip-flop instance names are ff0, ff1, ff2, and ff3, enter

```
dc_shell> set_fsm_state_vector {ff0 ff1 ff2 ff3}
```

In this example, if the FSM has fewer than 16 legal states, you should use the `set_fsm_encoding` command to limit the legal states (see the next section on state encodings).

For more information about the `set_fsm_state_vector` command, see the man page.

State Encodings

FSM encodings define all the legal state bit encodings of an FSM, along with the symbolic names for these states. These encodings determine which state vector bit patterns represent legal states (see the preceding section) and which bit patterns can be treated as don't care states (not actual states of the FSM). The presence of don't care states increases the chances that the tool can achieve improved optimization results.

Conversely, without state encodings, all states are treated as legal in the sense that all states must be realizable in the compiled result. In this case, improved optimization results are not likely.

For most HDL design files, as well as state table files, Design Compiler can derive the state encodings. If the state encodings cannot be derived by the tool and you want to take advantage of don't care states during optimization, you can use the `set_fsm_encoding` command to establish the encodings.

For more information about the `set_fsm_encoding` command, see the man page.

State Encoding Styles

The Design Compiler tool can use one of four state encoding styles—one-hot, binary, gray, and auto—to assign state encodings to legal states *during optimization* (after FSM extraction). The different encoding styles can lead to very different FSM optimization results, which gives you some flexibility in trying to improve the QoR of a compiled FSM.

The one-hot, binary, and gray encoding styles generate state assignments that override all prior state encodings and can lead to a redefinition of the state vector. The auto encoding style assigns state encodings only to unencoded states; it does not override previously specified state encodings.

The one-hot encoding style generates codes with a bit length equal to the number of states in the FSM. Each state is represented by a unique bit pattern that has a 1 in only one bit position and 0s in all other bit positions, so there are as many flip-flops required as there are states. This encoding style simplifies the combinational logic of the FSM and usually yields the fastest machine. However, you can expect the area of the FSM to increase.

The binary and gray encoding styles assign state codes to *ordered* states, based on a binary or gray numbering sequence. A specific state order is required. Either it is derived by the tool from the HDL input file or from the state table (if the optional `.encoding` section is present), or it is defined by you, using the `set_fsm_order` command. These encoding styles tend to require the least number of flip-flops — $\log_2(\text{number of states})$ —and can often lead to reduced area QoR.

The auto encoding style generates state codes in a seemingly random fashion that best reduces the complexity of the combinational logic of the FSM while using a minimum number of encoding bits. This style can lead to the best overall QoR.

In the automatic FSM flow, state reencoding occurs automatically as part of the FSM optimization phase. You do not have to manually remove state encodings or choose the state encoding style. In this case, the tool chooses the encoding style that achieves optimum QoR. You can, however, specify the encoding style by using the `set_fsm_encoding_style` command. For example, to use the binary encoding style instead of having the tool select the style, enter

```
dc_shell> set_fsm_encoding_style binary
```

before you compile the design.

For more information about the `set_fsm_order` and `set_fsm_encoding_style` commands, see the man pages.

Completely and Incompletely Specified Finite State Machines

Finite state machines do not always have their state transition behavior, outputs, or state encodings completely specified. So-called *don't care* conditions can be substituted for certain kinds of incomplete specification.

FSMs can have the following sets of don't care conditions:

- Input conditions for which the next state of the machine is unspecified. These input conditions constitute the next-state don't care set and are obtained by the tool from the state table description.
- Output conditions for which the value for certain outputs is unspecified. These output conditions constitute the output don't care set and are obtained by the tool from the state table description.
- State codes not used in the particular encoding of the FSM. These state codes constitute the encoding don't care set and are automatically derived by the tool after the state encodings are known.

An FSM for which the state transition behavior is specified for all possible input conditions is a completely specified machine, irrespective of any don't care conditions that might apply to its outputs and state encodings. But an FSM for which the state transition behavior is not specified for all possible input conditions is an *incompletely specified* machine.

The next-state don't care set is a sequential don't care set. The FSM's behavior is undefined and can transition to any state in the machine, including invalid states.

If an incompletely specified FSM description is read into Design Compiler, the following message is displayed during compilation:

```
Warning: In design name, the next state is unspecified for  
some transitions. (FSM-104)
```

The output and encoding don't care sets are combinational don't care sets. They can be used during FSM compile to simplify the combinational logic of the machine.

Synthesizing Finite State Machines

The FSM optimization algorithms require that the FSM part of an input design file be described in Synopsys state table format or its equivalent internal data structure. In general, designs are not represented by state tables. Therefore, in most cases, it is necessary to *extract* the FSM portion of the circuit from the initial design, convert the extracted logic to a state table, and replace the original FSM logic with the state table description before the entire design is compiled. These steps are carried out automatically through a series of `dc_shell` commands.

Commands Supported in the DC Ultra Automatic Flow

In the DC Ultra automatic FSM flow, the processes are transparent, including extracting the FSM and generating an intermediate state table representation. In most cases, you issue only the `read_file`, `compile`, and `report_fsm` commands. (Other commands are available if needed.) After optimization, the entire design is a mapped, technology-dependent database file. The FSM objects in the database file are marked with the finite state machine attributes.

In addition to the `read_file`, `compile`, and `write` commands, the commands supported in the automatic flow include the following:

- `set_fsm_state_vector`
- `set_fsm_encoding`
- `group -fsm`
- `set_fsm_order`
- `set_fsm_encoding_style`
- `set_fsm_preserve_state`
- `set_fsm_minimize`
- `report_fsm`

Usually, you do not need to use the additional commands. However, if Design Compiler cannot recognize the FSM part of an input design and you know *exactly* both the state vector and state encodings for the design, you can provide this critical information by using the `set_fsm_state_vector` and `set_fsm_encoding` commands. Also, you can investigate the compiled design's QoR for different state assignments by using the `set_fsm_encoding_style` command.

For more information about all these commands, see the man pages.

Finite State Machine Design File Requirements

For Design Compiler to be able to extract and optimize an FSM either automatically or manually, the initial design must meet the requirements described in [Table 8-1](#).

Table 8-1 FSM Requirements

Item	Description
Ports	All ports of the initial design must be input ports or output ports. Inout ports are not supported.
Function	Only one FSM design per module (Verilog) or entity (VHDL) is recommended. If multiple FSMs are present, only one is extracted each time you compile. It is not possible to predict which FSM will be extracted. State variables cannot drive a port. State variables cannot be indexed.
Combinational feedback loops	Combinational feedback loops are not supported, although combinational logic that does not depend on the state vector is accurately represented.
Clocks	FSM designs can include only a single clock and an optional synchronous or asynchronous reset signal that resets to the initial state. These signals must connect only to each state vector element (flip-flop). The clock signal must have the same rising or falling sense for all state vector elements.

Note:

Design Compiler automatically determines the reset state of the FSM during the extraction process by noting how the asynchronous reset signal is connected to the set or reset pins of each state vector flip-flop. The extracted encoding of the asynchronous reset state must be a valid state of the machine.

In addition, for Verilog design files, the following restrictions must be observed in the file:

- A state register cannot occur directly on the right side of an assignment except when the operand of either the “==” or “!=” operator.
- A state register must infer flip-flops (not latches).
- A state register cannot be a port.

DC Ultra Automatic Methodology

DC Ultra FSM optimization is enabled by default when you use the `compile_ultra` command.

You must also ensure that the `fsm_auto_inferring` variable is set to true (the default is false).

Note:

If clock gating is enabled, the automatic FSM flow is automatically disabled.

You can also set the following variables to true:

- `fsm_enable_state_minimization`

The default is false. Use the default if you want to be able to verify the compiled design. State minimization removes any redundant states and is automatically performed if this variable is true.

Note:

If state minimization does change the number of states in the FSM, the Formality tool cannot verify the compiled design.

- `fsm_export_formality_state_info`

The default is false. If you set this variable to true, the state encoding information before and after compile is saved in a `.ref` file and an `.imp` file, respectively. These files are used in the Formality verification process.

How Design Compiler Processes a Finite State Machine in the DC Ultra Automatic Flow

Design Compiler processes an FSM input file in two phases that are transparent to the user: a read phase followed by a compile phase.

The read phase consists of three steps:

1. Read in the input HDL design or state table `.st` file.
2. Autodetect the FSM registers (flip-flops) of the input design.
3. Mark the FSM state vector and state encoding attributes on the design.

The compile phase consists of the following steps:

1. Autopartition the FSM from the input design.

This step creates a new hierarchy that contains the FSM combinational logic and state vector registers.

2. Autoextract the FSM from the newly created hierarchy.

This step creates an intermediate state table format that resembles the format of a state table .st file but is an internal data structure.

3. Check for a user-specified state encoding style, and, if not specified, autoselect the style that might produce the best QoR.

The selected encoding style affects state assignment. Available encoding styles include one-hot, binary, gray, and auto.

4. Perform state minimization, if enabled.
5. Perform state assignment (also referred to as state encoding) on the FSM hierarchy.
6. Generate the Design Compiler internal data structure from the FSM logic netlist according to the state assignment.
7. Flatten the newly created FSM hierarchy.
8. Continue with the standard Design Compiler Ultra optimization steps.
9. Output the technology-mapped database file with the FSM objects marked with the FSM attribute.

The Finite State Machine DC Ultra Automatic Flow

Figure 8-2 shows a simplified DC Ultra automatic FSM flow for both HDL input design files and Synopsys state table files. In most cases, you can use this flow.

Figure 8-2 DC Ultra Automatic Finite State Machine: Simple Flow

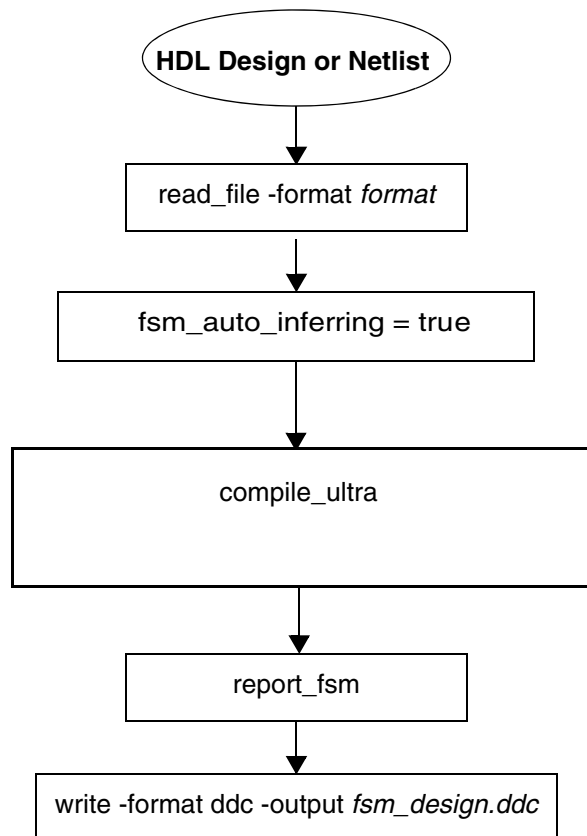
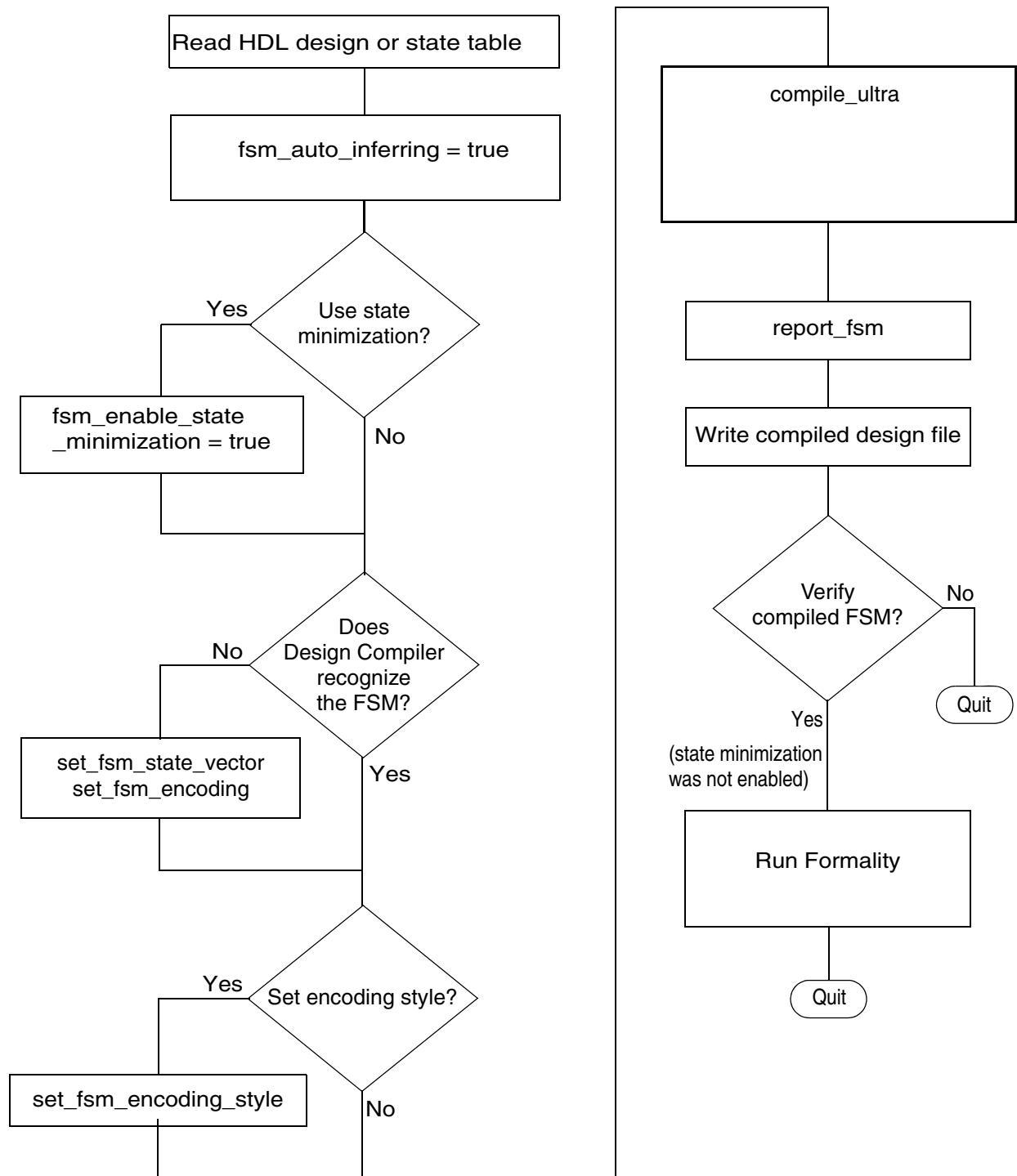


Figure 8-3 shows the complete DC Ultra automatic FSM flow for both HDL input design files and Synopsys state table files. In this flow, the various optional commands are shown. Depending on the design you are trying to compile, you might have to use one or more of these commands. Also, optionally, you might want to try to improve the QoR for your compiled design by using some of these commands.

Figure 8-3 DC Ultra Automatic Finite State Machine: Complete Flow



For most HDL input files, you do not specify the state vector and state encodings. However, if Design Compiler cannot automatically recognize and extract the FSM logic, you must use the `set_fsm_state_vector` and, optionally, the `set_fsm_encoding` commands. To experiment with the QoR, you might want to try different encoding styles by using the `set_fsm_encoding_style` command.

After the design is compiled, you can verify the FSM by using the Formality tool.

For more information about design verification, see [Chapter 13, “Verifying Functional Equivalence”](#) or the *Formality User Guide*.

Verifying a Finite State Machine

If you did not enable state minimization (`fsm_enable_state_minimization` is true), you can use the Formality tool to verify your compiled FSM design. For information about design verification, see [Chapter 13, “Verifying Functional Equivalence”](#) and the *Formality User Guide*.

Creating Finite State Machine Reports

The `report_fsm` command creates an FSM report that includes the following information:

- Name of the clock signal and its sense
- Name of the optional asynchronous reset signal, its sense, and its state name
- Encoding bit length and encoding style
- State vector flip-flop names and ordering
- A list of state names and encodings in state order
- Preserved states and merged (equivalent) states

To obtain an FSM report, enter

```
dc_shell> report_fsm
```

For more information about the FSM report, see the man page.

The following is a typical FSM report:

```
*****
Report : fsm
Design : BUS_ARBITRATOR
Version: v2001.08
Date   : Tues July 17 2001
*****
Clock           : CLK           Sense: rising_edge
Asynchronous Reset: Unspecified

Encoding Bit Length: 3
Encoding style      : auto

State Vector: { FF2 FF1 FF0 }

State Encodings and Order:

Grant_A      : 001
Wait_A       : 011
Timeout_A1   : 111
Grant_B      : 010
Wait_B       : 110
Timeout_B1   : 101

Preserved States: Grant_A

Merged States: None
```

9

Sequential Mapping

Sequential mapping phase consists of two steps: register inferencing and technology mapping. Synopsys uses the term register for both edge-triggered registers and level-sensitive latches.

Register inferencing is the process by which the RTL description of a register is translated into a technology-independent representation called a SEQGEN. SEQGENs are created during elaboration and are usually mapped to flip-flops during compile. Technology mapping is the process by which a SEQGEN is mapped to gates from a specified target logic library. It is performed when you use the `compile_ultra` or `compile` command. Before you read this chapter, read the [“Optimization Flow” on page 1-9](#) to understand how sequential mapping fits into the overall compile flow.

This chapter contains the following sections:

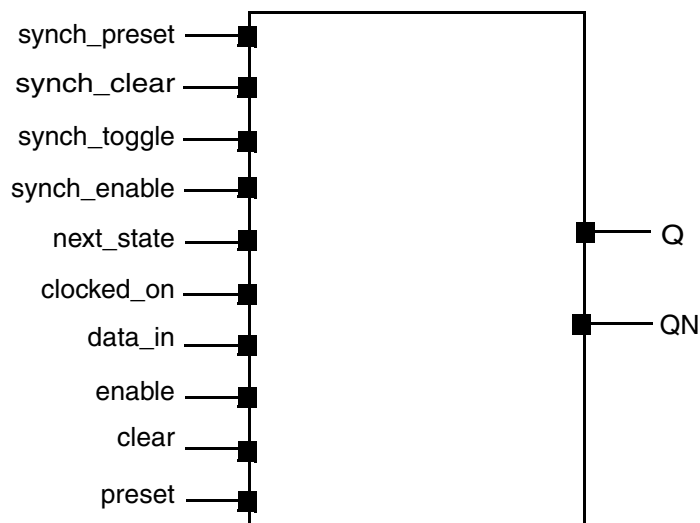
- [Register Inference](#)
- [Directing Register Mapping](#)
- [Specifying the Default Flip-Flop or Latch](#)
- [Reporting Register Types](#)
- [Unmapped Registers in a Compiled Design](#)
- [Automatically Removing Unnecessary Registers](#)
- [Merging Equal and Opposite Registers](#)

- [Inverting the Output Phase of Sequential Elements](#)
- [Mapping to Falling-Edge Flip-Flops](#)
- [Resizing Black Box Registers](#)
- [Preventing the Exchange of the Clock and Clock Enable Pin Connections](#)
- [Mapping to Registers With Synchronous Reset or Preset Pins](#)
- [Performing Test-Ready Compile](#)
- [Using Register Replication to Solve Timing QoR, Congestion, and Fanout Problems](#)

Register Inference

When HDL Compiler reads in a Verilog or VHDL RTL description of the design, it translates the design into a technology-independent representation (GTECH). In GTECH, both registers and latches are represented by a SEQGEN cell, which is a technology-independent model of a sequential element as shown in [Figure 9-1](#). SEQGEN cells have all the possible control and data pins that can be present on a sequential element.

Figure 9-1 Generic SEQGEN Cell



[Table 9-1](#) lists the pins of a SEQGEN cell. Only a subset of these pins is used depending on the type of cell that is inferred. Unused pins are tied to zero.

Table 9-1 Pins of a SEQGEN Cell

Direction	Name	Description	Cell type
Input	clocked_on	clock	flip-flop
	next_state	synchronous data	flip-flop
	data_in	asynchronous data	latch
	synch_toggle	synchronous toggle	flip-flop
	synch_clear	synchronous reset	flip-flop
	clear	asynchronous reset	flip-flop or latch
	preset	asynchronous preset	flip-flop or latch
	synch_enable	synchronous enable	flip-flop
	asynch_enable	asynchronous enable	latch
Output	Q	non-inverting output	flip-flop or latch
	QN	inverting output	flip-flop or latch

Incorrect register inferencing results in incorrect technology mapping. One way to examine the type of register inferred is to examine the register inference reports in HDL Compiler. Set the following variable in HDL Compiler to generate additional information on inferred registers:

```
set hdl_in_report_inferred_modules verbose
```

[Example 9-1](#) shows an HDL Compiler inference report for a D flip-flop with a synchronous preset control.

Example 9-1 Inference Report

```
=====
| Register Name | Type | Width | Bus | MB | AR | AS | SR | SS | ST |
=====
| Q_reg | Flip-flop | 1 | N | N | N | N | N | Y | N |
=====
Sequential Cell (Q_reg)
Cell Type: Flip-Flop
Multibit Attribute: N
```

```
Clock: CLK
Async Clear: 0
Async Set: 0
Async Load: 0
Sync Clear: 0
Sync Set: SET
Sync Toggle: 0
Sync Load: 1
```

The inference report can help troubleshoot issues by indicating the type of register inferred (latch or flip-flop, single or multibit) and the control signals inferred for that register. The report shows the names of the nets connecting the pins on the SEQGEN elements in GTECH. In most cases, these control pins are tied to zero (inactive). If you are attempting to infer a synchronous or asynchronous reset or preset and it does not correctly appear in the inference report, check for issues in the specification of the register at the RTL level.

Correct inferencing of synchronous and asynchronous reset or preset control signals in the RTL results in connections to the `synch_set`, `synch_clear`, `asynch_set`, and `asynch_clear` pins on the SEQGEN GTECH cell. During technology mapping, Design Compiler uses the SEQGEN as the starting point for mapping. The sequential mapper checks the connections to the pins and the information present in the library cell descriptions when it maps to a logic library register.

Incorrect register inferencing or incomplete library information can produce unexpected results in mapping to the reset or preset pins of the register.

Design Compiler does not infer synchronous resets by default. To indicate to the tool which signals should be treated as synchronous resets, use the `sync_set_reset` Synopsys compiler directive in Verilog source files or the corresponding `sync_set_reset` Synopsys attribute in VHDL source files. HDL Compiler then connects these signals to the `synch_clear` and `synch_preset` pins on the SEQGEN in order to communicate to the mapper that these are the synchronous control signals and they should be kept as close to the register as possible. For information on inference of these signals, see the HDL Compiler documentation. For information on how Design Compiler maps these signals, see [“Mapping to Registers With Synchronous Reset or Preset Pins” on page 9-16](#).

The correct mapping of asynchronous reset or preset registers requires that these registers be correctly described in the RTL and that corresponding cells exist in the logic library. As long as you follow the recommended coding guidelines for coding asynchronous registers, no special compiler directives are needed in order to infer asynchronous set or reset signals. For more information on inference of these signals, see the HDL Compiler documentation.

Directing Register Mapping

You can control register mapping in the following ways:

- Use the `set_register_type` command. See [“Specifying the Default Flip-Flop or Latch.”](#)
- Use exact mapping (with the `-exact_map` option).

Use the `-exact_map` option of the `compile_ultra` command or `compile` command to restrict the mapping to sequential cells with simple behavior (synchronous set and reset, synchronous toggle, synchronous enable, asynchronous set and reset, and asynchronous load and data). When you use the `-exact_map` option, sequential mapping does not try to encapsulate combinational logic originally outside the generic sequential element (SEQGEN) into the sequential cell.

- Use test-ready compile (`compile_ultra -scan` or `compile -scan`). See [“Performing Test-Ready Compile”](#) on page 9-20.

Specifying the Default Flip-Flop or Latch

The `set_register_type` command specifies the default flip-flop or latch library cell type for some or all registers in the current design or current instance. A flip-flop type is represented by an example flip-flop; any flip-flop that has the same sequential characteristics as the specified flip-flop is considered to be of that type.

Use the `set_register_type` command to direct Design Compiler to infer a particular flip-flop or latch.

Note:

This mechanism for directing register types is not needed if you write the HDL to directly infer the correct register type.

You can specify a latch, a flip-flop, or both.

For example,

To set the default flip-flop type to FFX and the default latch type to LTCHZ, enter

```
dc_shell> set_register_type -flip_flop FFX -latch LTCHZ
```

For more information, see the `set_register_type` man page.

Reporting Register Types

You can see the current default register type specification for the design and for cells.

Reporting the Register Type Specifications for the Design

The `report_design` command lists the current default register type specifications.

```
dc_shell> report_design

*****
Report : design
Design : DESIGN
Version: Y-2006.06
Date   : Mon May 1 16:52:43 2006
*****

. . .
Flip-Flop Types:
    Default: FFX, FFXHP, FFXLP
```

Reporting the Register Type Specifications for Cells

The `report_cell all_registers` command lists the current register type specifications for cells.

The syntax is

```
report_cell [all_registers]
```

The following example shows a sample report.

```
*****
Report : cell
Design : reg_type
Version: Y-2006.06
Date   : Mon May 1 2006
*****

Attributes:
    n - noncombinational
...
```

Cell	Reference	Library	Area	Attributes
ffa	FFY	MY_LIB	9.00	1,n
ffb	FFY	MY_LIB	9.00	1,n
ffc	FFY	MY_LIB	9.00	1,n
ffd	FFY	MY_LIB	9.00	1,n

Total 4 cells 36.00

Flip-Flop Types:
1 - Exact type FFY

Unmapped Registers in a Compiled Design

When Design Compiler fails to find a match for a register in the available target logic library, it issues the following warning in the compile log:

```
Warning: Target library contains no replacement for register
'Q_reg' (**FFGEN**). (TRANS-4)
```

In addition, the compiled gate-level netlist has the following type of cell:

```
\**FFGEN** Q_reg ( .next_state(D), .clocked_on(CLK),
.force_00(1'b0), .force_01(N0), .force_10(1'b0),
.force_11(1'b0), .Q(Q) );
```

****FFGEN**** is a model of the register functionality, similar to the SEQGEN. It is not a cell that you find in any logic library. Check the names of the cells mapped to ****FFGEN****; these are the cells for which Design Compiler could not find a match in the logic library. This behavior usually occurs when asynchronous registers or latches are inferred in the RTL but a cell with the corresponding asynchronous functionality is not available for mapping in the specified target libraries.

Check the failing registers to see the types of registers that you are attempting to infer:

- Are you inferring positive-edge or negative-edge clocked registers?
- What sets of asynchronous control pins are being inferred (reset only, preset only, both reset and preset)?
- Are you performing power gating on these cells (using retention registers)?
- Are you using a test-ready compile (`compile_ultra -scan` or `compile -scan`)?

Make sure that your target library has cells with all the features that you are attempting to use for register mapping. Verify that you have not disabled the use of the required cells with the `set_dont_use` command. Use the following command to check the `set_dont_use` settings:

```
write_environment -environment_only
```

Automatically Removing Unnecessary Registers

During sequential mapping, Design Compiler can save area significantly by automatically detecting and removing unconnected registers and constant registers.

Removing Unconnected Registers

During optimization, Design Compiler deletes registers having outputs that do not drive any loads. The combinational logic cone associated with the input of the register can also be deleted if the cell is not used elsewhere in the design. Register outputs can become unconnected due to redundancy in the circuit or as a result of constant propagation. In some designs where the registers have been instantiated, the outputs might already be unconnected.

You can preserve such registers if you need to maintain consistency between the compiled design and the HDL source or for other design reasons, such as in the case of instantiated cells. To direct Design Compiler to preserve the registers, set the `compile_delete_unloaded_sequential_cells` variable to false before you compile. The default is true. When this variable is set to false, a warning message appears during compilation, indicating the presence of unloaded registers.

```
Warning: In design 'design_name', there are sequential cells
not connected to any load. (OPT-109)
Information: Use the 'check_design' command for more
information about warnings. (LINT-99)
```

You can use the `check_design` command after compile to identify cell instances that have unconnected outputs.

In cases where a feedback loop exists with no path from any section of the loop to a primary output, no warning appears because the output of the register is theoretically connected. When the `compile_delete_unloaded_sequential_cells` variable is set to true, such cells are optimized away. Setting the value to false retains the cells and the logic cone associated with the inputs to the cells.

Eliminating Constant Registers

Certain registers in a design might never change their state because they have constant values on one or more input pins. These constant values can either be directly at the input or result from the optimization of fanin logic that eventually leads to a constant input of the register. Eliminating such registers can improve area significantly.

During compile, Design Compiler performs constant propagation to automatically find and replace such sequential elements with a constant. This type of optimization is controlled by

setting the `compile_seqmap_propagate_constants` variable to true (default value). When this type of optimization is performed, Design Compiler issues an informational message in the log file when it removes a constant register in the design:

```
Information: The register 'Z_reg' is a constant and will be
removed. (OPT-1206)
```

Table 9-2 lists cases in which a sequential element can be eliminated.

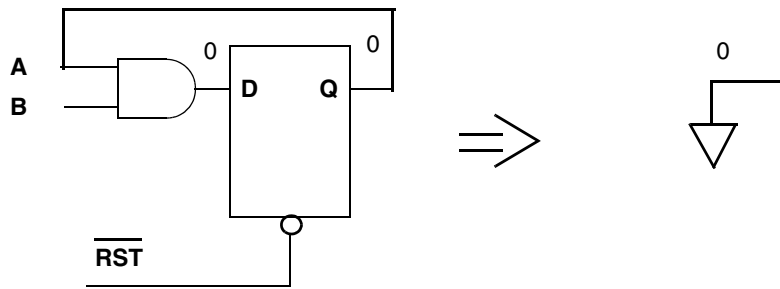
Table 9-2 Cases for Which Sequential Elements Are Eliminated

Type of register	Data	Preset	Reset
Simple, constant data	1 or 0		
Preset and constant data 1	1	X	
Constant preset	X	1	
Reset and constant data 0	0		X
Constant reset	X		1
Preset, reset, and constant data 1; reset always inactive	1	X	0
Preset, reset, and constant data 0; preset always inactive	0	0	X

Additionally, Design Compiler removes sequential elements for which the logic leading to a constant value is particularly complex. That is, it attempts to identify more complex conditions leading to a register input. It analyzes each sequential element to determine its reset state. If a known reset state can be determined, Design Compiler checks whether the sequential element can switch state after it has reached its known reset state. If the sequential element cannot escape its reset state, Design Compiler replaces it with a constant equivalent to its reset state.

Figure 9-2 illustrates such a case. A feedback path leads from the Q pin of the register back to the D input; the register cannot escape its reset state.

Figure 9-2 Constant Output Sequential Element That Cannot Escape Its Reset State



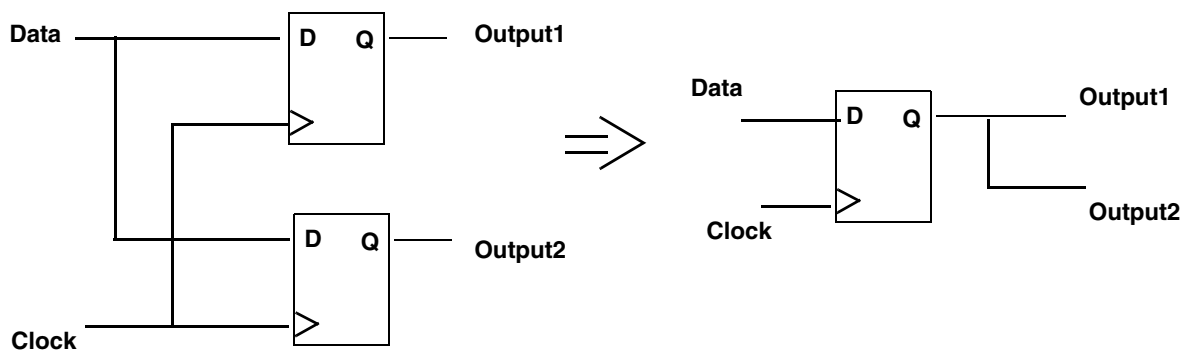
Note:

Constant propagation occurs for both unmapped and mapped designs, when the `compile_seqmap_propagate_constants` variable is set to true. However, the best results are obtained when constant propagation is enabled during the initial sequential mapping from an unmapped design.

Merging Equal and Opposite Registers

By default, Design Compiler identifies and merges equal and opposite registers. Two registers that are equal in all states that can be reached from the reset state can be replaced with a single register driving both sets of loads. The same principle applies to registers that are opposite to each other. In addition to removing the registers, this type of optimization enables equal and opposite information to propagate beyond the registers so that subsequent logic can be optimized in the context of equal and opposite relationships. Equal and opposite register merging is shown in [Figure 9-3](#).

Figure 9-3 Merging Equal and Opposite Registers



Design Compiler issues the following message when it merges registers:

```
Information: In design 'test', the register 'u2/op_reg' is removed
because
it is merged to 'u2/op1_reg'. (OPT-1215)
```

To disable register merging on specific cells or blocks, set the `set_register_merging` command to `false` on those cells or blocks. If you want to prevent register merging on all registers in your design, set the `compile_enable_register_merging` variable to `false`. If you set the `compile_enable_register_merging` variable to `false`, you cannot enable any register merging, that is, the `compile_enable_register_merging` variable setting takes precedence over the `set_register_merging` command.

To make sure that the `register_merging` attribute is set on a cell or design, use the `report_attribute` command. For example,

```
dc_shell> report_attribute u1/op_reg
```

Design	Object	Type	Attribute Name	Value
test	u1/op_reg	cell	register_merging	false
test	u1/op_reg	cell	is_a_generic_seq	true
test	u1/op_reg	cell	ff_edge_sense	

Inverting the Output Phase of Sequential Elements

The `compile_seqmap_enable_output_inversion` variable controls whether the `compile` command allows sequential elements to have their output phase inverted. Note that the `compile_seqmap_enable_output_inversion` variable does not have any effect on the `compile_ultra` command; to control sequential output inversion for the `compile_ultra` command, use the `-no_seq_output_inversion` option.

In certain cases, you might be inferring a register that has one type of asynchronous control but your library has the opposite type of pin. For example, you might be inferring an asynchronous set but your library has only registers with asynchronous clear pins.

During mapping, Design Compiler issues the following warning message:

```
Warning: Target library contains no replacement for register
'Q_reg' (**FFGEN**). (TRANS-4)
```

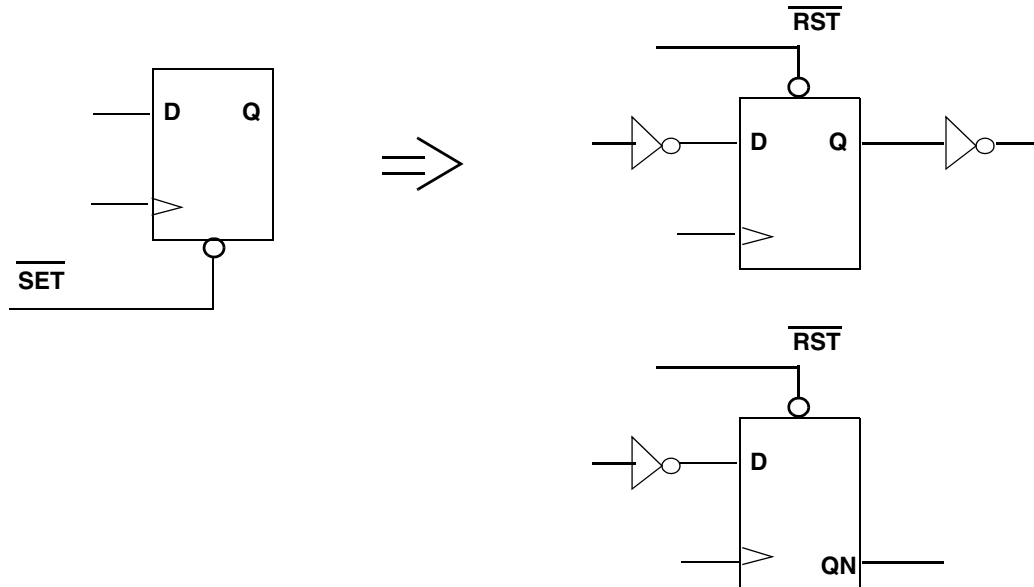
In such cases, you can allow Design Compiler to map to the opposite type of register and invert all the data inputs and outputs by using the

`compile_seqmap_enable_output_inversion` variable to as follows:

```
set compile_seqmap_enable_output_inversion true
```

Figure 9-4 shows an example of output inversion transformations.

Figure 9-4 Output Inversion Transformations



Note:

Information about inverted registers is written to the SVF file. The following informational message appears in the log file to remind you that you must include the SVF file in Formality when verifying designs with output inversion:

Information: Sequential output inversion is enabled. SVF file must be used for formal verification. (OPT-1208)

Mapping to Falling-Edge Flip-Flops

Falling-edge flip-flops are inferred by HDL Compiler by referencing the falling edge of the clock in the process describing the register. The resulting SEQGEN shows an inversion of the clocked_on pin as shown in Figure 9-5. HDL Compiler also sets attributes so that the register is mapped to a flip-flop with an inverted clock input as shown in Figure 9-6 on page 9-15. You do not have to define any constraints to enable mapping to falling-edge flip-flops.

Example

For the following Verilog design, HDL Compiler creates the SEQGEN shown in Figure 9-5.

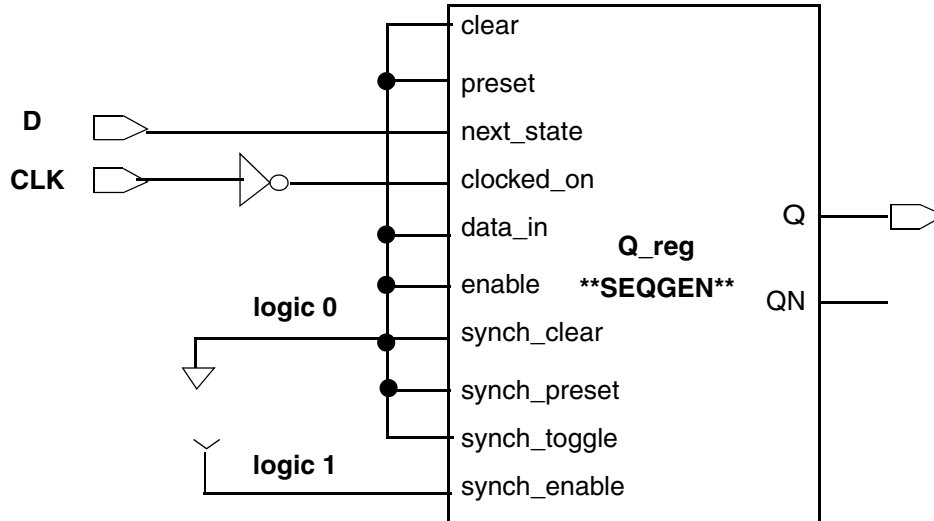
```
module dff_inv_clk (D,CLK,Q);
input D,CLK;
output reg Q;
always @ (negedge clock)
```

```

Q = D;
endmodule

```

Figure 9-5 SEQGEN Created by HDL Compiler



Note:

The inverter on the clock net in the GTECH is only used by the tool to represent an inverted clock for mapping the registers. It does not imply that an inverter is put on the clock net.

Example 9-2 shows the inference report.

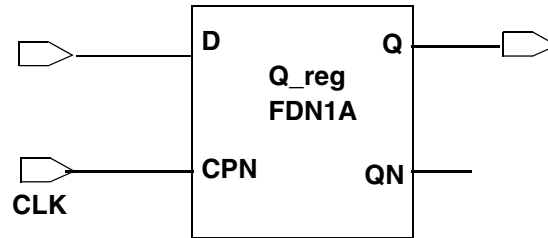
Example 9-2 Inference Report

```

=====
| Register Name | Type      | Width | Bus | MB | AR | AS | SR | SS | ST |
=====
| Q_reg        | Flip-flop | 1     | N   | N  | N  | N  | N  | N  | N  |
=====
Sequential Cell (Q_reg)
  Cell Type: Flip-Flop
  Multibit Attribute: N
  Clock: CLK'
  Async Clear: 0
  Async Set: 0
  Async Load: 0
  Sync Clear: 0
  Sync Set: 0
  Sync Toggle: 0
  Sync Load: 1

```

Given a library with both positive edge and negative edge-triggered flip-flops, Design Compiler creates the optimized negative edge-triggered design as shown in Figure 9-6.

Figure 9-6 Register Mapped To Logic Library

Resizing Black Box Registers

Design Compiler supports the `user_function_class` attribute for cells that cannot be functionally modeled for synthesis. Design Compiler treats black box cells with the same `user_function_class` attribute and the same number of pins as functionally equivalent. The tool can resize such cells by using other cells with the same `user_function_class` attribute value, as long as timing arcs to the output pins of the cells are provided in the target logic library. This capability works for both combinational and sequential library cells.

If this attribute has not already been set on your black box registers in the library source (.lib) file, you can set it by using the `set_attribute` command. During register-sizing optimizations, registers with the attribute setting can be exchanged with other registers having the same `user_function_class` attribute. For example,

```
set_attribute [get_lib_cells my_lib/DFFX*]\
               user_function_class DFFX -type string
```

Information: Attribute 'user_function_class' is set on 4 objects. (UID-186)

```
my_lib/DFFX1 my_lib/DFFX2 my_lib/DFFX4 my_lib/DFFX8
```

Note that by default, Design Compiler does not resize integrated clock-gating cells. Such cells have the `clock_gating_integrated_cell` attribute set on the library cell and can only be sized by Power Compiler. Use the `identify_clock_gating` command to identify existing integrated clock-gating cells in the design.

Preventing the Exchange of the Clock and Clock Enable Pin Connections

Some libraries have flip-flops with both clock and clock enable pins. Although Design Compiler can successfully map to both of these pins, incomplete information in the library can lead to situations in which the clock and clock enable pins are exchanged during mapping.

In the library description of the flip-flop (.lib file), all pins included in the `clocked_on` attribute are treated as clocks by default. The `pin_func_type` attribute alone is not sufficient to distinguish a clock enable pin from a clock pin. You must also set the clock attribute to false on clock enable pins to prevent these pins from being treated as clocks by the sequential mapper.

Clock nets are distinguished from clock enable nets in the design based on clock constraints propagated to these nets. The following simplified library example shows how you can distinguish the clock enable pin (CE in the example) from the clock pin (CLK in the example) by specifying the `pin_func_type` attribute as `clock_enable` and also by setting the clock attribute to false on this pin:

```
cell (DFFCE) {
ff (IQ, IQN) {
next_state : "D" ;
clocked_on : "CLK & CE" ;
}
pin (CLK) {
direction : input;
clock : true;
}
pin (CE) {
direction : input;
pin_func_type : clock_enable;
clock : false ;
}
...
}
```

Mapping to Registers With Synchronous Reset or Preset Pins

Before you read this section, see [“Register Inference” on page 9-3](#) for information on inference of synchronous reset or preset pins. Flip-flops with synchronous resets or presets are mapped in one of the following ways:

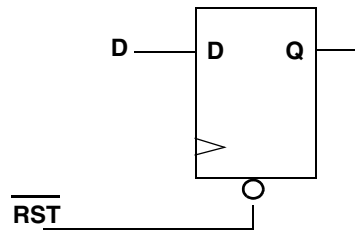
- If your library has registers with synchronous reset (or preset) pins, the reset (or preset) net is connected to the reset (or preset) pin of a register with a dedicated reset pin.
- If your library does not have any registers with synchronous reset (or preset) pins, the tool adds extra logic to the data input to generate the reset (or preset) condition on a

register without a reset (or preset) pin. In these cases, Design Compiler attempts to map the logic as close as possible to the data pin to minimize X-propagation problems that lead to synthesis/simulation mismatches.

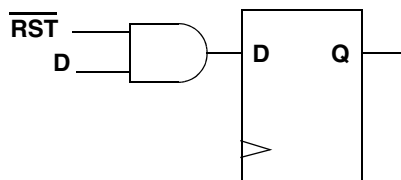
Figure 9-7 shows examples of mapping to registers with and without a synchronous reset pin.

Figure 9-7 Mapping To Registers With and Without Synchronous Reset Pins

Synchronous Reset Using a Reset Pin



Synchronous Reset Using a Gate on the Data Pin



If your library does have registers with synchronous reset (or preset) pins, it is still important to use the `sync_set_reset` pragma in your RTL so that Design Compiler can distinguish the reset (or preset) signals from other data signals and connect the reset signal as close to the register as possible.

Note:

Synchronous reset and preset signals are not inferred for level-sensitive latches. A synchronous reset or preset coding style on a latch always results in combinational logic on the data signal even if the library has latches with synchronous reset or preset pins.

Even if your library does contain registers with synchronous reset or preset pins, these registers are used for mapping. However, you must ensure that the synchronous pins in the library have the `nextstate_type` attribute correctly defined for each of these pins to correctly connect the pins.

The `nextstate_type` attribute is predefined for use in the .lib file as follows:

```
nextstate_type : data | preset | clear | load | scan_in |
scan_enable;
```

The `nextstate_type` attribute is used by the sequential mapper in order to distinguish between the different types of synchronous input pins. Without this information, the synchronous data, reset, and clear pins are all treated as synchronous data pins and may be exchanged during mapping.

[Table 9-3](#) describes the attribute values and their corresponding short integer value.

Table 9-3 Values Used for the `nextstate_type` Attribute

Enum value used in the .lib file	Short type value in Design Compiler	How the pin is identified
data	0	Synchronous data pin (the default)
preset	1	Synchronous preset pin
clear	2	Synchronous clear pin
load	3	Synchronous load pin
scan_in	4	Synchronous scan in pin
scan_enable	5	Synchronous scan enable pin

[Example 9-3](#) shows how the `nextstate_type` attribute (in **bold** type) is specified in the library for a synchronous reset register. Other necessary attributes have been omitted for the sake of clarity.

Example 9-3 Specifying the `nextstate_type` Attribute

```
cell (DFFSRST) {
  ff (IQ, IQN) {
    next_state : "RN D" ;
    clocked_on : "CLK" ;
  }
  pin (CLK) {
    direction : input ;
    clock : true ;
  }
  pin (D) {
    direction : input ;
    nextstate_type : data;
  }
  pin (RN) {
    direction : input ;
    nextstate_type : clear;
  }
  pin (Q) {
    direction : output ;
    function : "IQ" ;
  }
}
```

```

}
} /* end of cell DFFSRST

```

Note how the `nextstate_type` attribute has been added to the pin groups for the data and synchronous reset pins. Pins included in the `next_state` attribute have a default `nextstate_type` of data unless otherwise specified. This setting might result in the swapping of the data and synchronous reset or preset pins if the `nextstate_type` attribute is not assigned.

You should therefore check to ensure that the `nextstate_type` attribute has been correctly added to the synchronous input pins of registers with synchronous reset or preset pins in your library.

If you are compiling the `.lib` file using Library Compiler, pay special attention to the following types of warnings for your synchronous set or reset registers:

```

Warning: Line 5905, The 'DFFSRST' cell is missing the
"nextstate_type" attribute for some input pin(s) specified
in 'next_state' of its ff/ff_bank group. (LIBG-243)

```

You can examine the `.lib` file if it is available. Otherwise you can check the library by querying for the attribute in `dc_shell`. The following example shows that the attributes have been correctly defined in the library.

```

dc_shell> read_db mytechlib.db
dc_shell> get_attribute [get_lib_pin MYTECHLIB/DFFSRST/D] nextstate_type
0

dc_shell> get_attribute [get_lib_pin MYTECHLIB/DFFSRST/RN] nextstate_type
2

```

The following example shows that the attributes are not correctly defined in the library:

```

dc_shell> get_attribute [get_lib_pin MYTECHLIB/DFFSRST/D] nextstate_type
Warning: Attribute 'nextstate_type' does not exist on port
'D'. (UID-101)

```

Ideally, you should add the attributes to the library source `.lib` file. Otherwise, you can add the attributes to the library pins in Design Compiler prior to using the library for synthesis. Note that you use the short type integer value in Design Compiler to add the missing attributes. See [Table 9-3 on page 9-18](#) for the appropriate value to use for each input pin type.

```

set_attribute [get_lib_pins MYTECHLIB/DFFSRST/D] \
              nextstate_type 0 -type short

set_attribute [get_lib_pins MYTECHLIB/DFFSRST/RN] \
              nextstate_type 2 -type short

```

After updating the library in memory by using these commands, you can save an updated version of the library by using the `write_lib` command as follows:

```
write_lib -format db MYTECHLIB -output mytechlib.fixed.db
```

For more information, see the Library Compiler documentation.

If the reset or preset registers are not being correctly mapped, check the following:

- Have you used the `sync_set_reset` compiler directive in your RTL to identify the set or reset?
- Does the register inference report from show that synchronous set or reset has been correctly inferred?
- Does the library have the `nextstate_type` attribute defined for the synchronous input pins?

Performing Test-Ready Compile

Test-ready compile integrates logic optimization and scan replacement. During the first synthesis pass of each HDL design or module, test-ready compile maps all sequential cells directly to scan cells. The optimization cost function considers the impact of the scan cells themselves and the additional loading due to the scan chain routing.

For a test-ready compile, you specify a scan style by using the `test_default_scan_style` or the `set_scan_configuration -style` command. Include the `-scan` option to the `compile_ultra` command or `compile` command when compiling the design. This section describes the following topics:

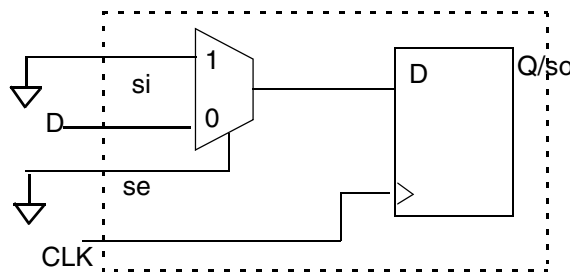
- [Overview of Test-Ready Compile](#)
- [Scan Replacement](#)
- [Selecting a Scan Style](#)
- [Mapping to Libraries Containing Only Scan Registers](#)
- [Mapping to the Dedicated Scan-Out Pin](#)
- [Automatic Identification of Shift Registers](#)

Overview of Test-Ready Compile

During test-ready compile, the tool replaces regular flip-flops with flip-flops that contain logic for testability. [Figure 9-8](#) shows an example of how a D flip-flop is replaced with a scan register during test-ready compile. This type of architecture, a multiplexed flip-flop, incorporates a two-input MUX at the input of the D flip-flop. The select line of the MUX enables two modes—functional mode (normal data input) or test mode (scanned data input). In this example, the scan-in pin is *si*, the scan-enable pin is *se*, and the scan-out pin, *so*, is shared with the functional output pin, *Q*.

Other architectures are supported: clocked scan, level-sensitive scan design (LSSD), and auxiliary-clock LSSD. The scan style dictates the appropriate scan cells to insert during optimization. You can change the default scan style by using the `test_default_scan_style` or the `set_scan_configuration -style` command. For more information, see the DFT Compiler documentation.

Figure 9-8 Example of a Scan Register Used During Test-Ready Compile



Note:

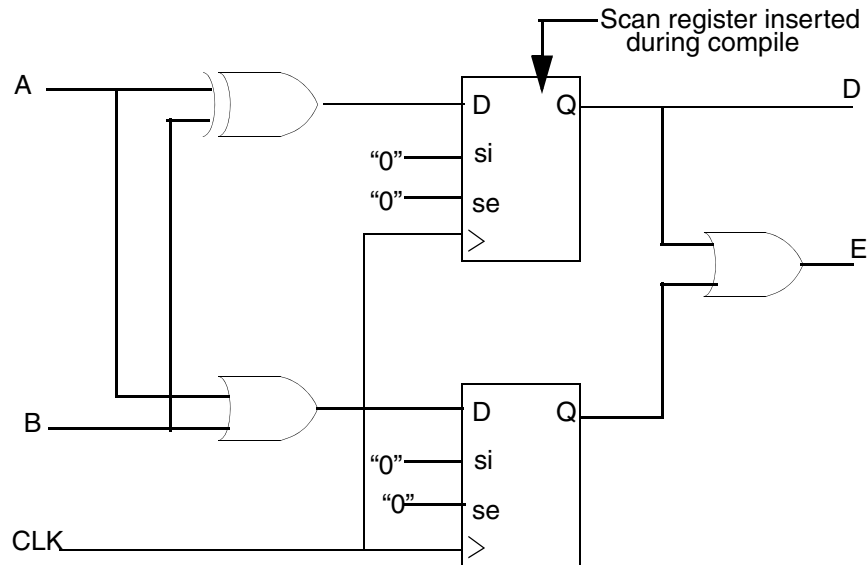
Scan connections are removed and modeled internally with pin or net capacitance. This results in improved QoR for both test-ready and scan-stitched designs. It also leads to better optimization opportunities, such as optimal cell sizing.

When you use test-ready compile, Design Compiler does the following:

- Maps all sequential cells directly to scan registers in your library
- Ties the test control pins to the appropriate state to enable functional mode

Figure 9-9 shows the result of a test-ready compile.

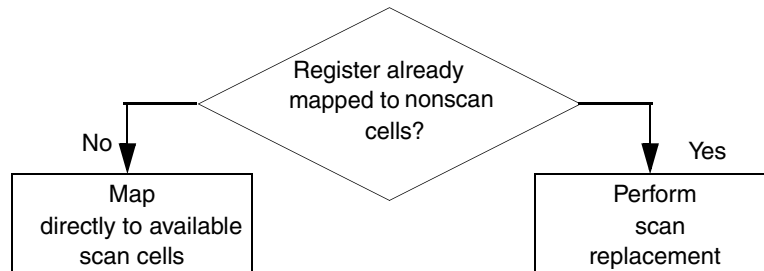
Figure 9-9 Result of Test-Ready Compile



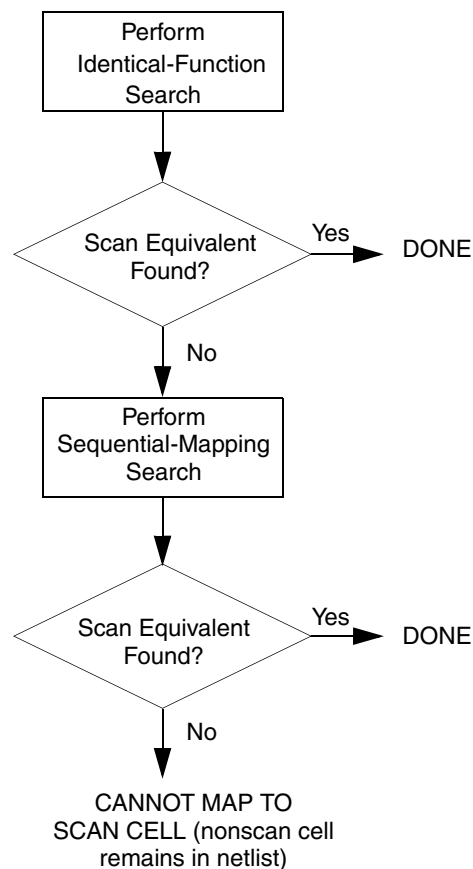
After test-ready compile, you have a design that contains unrouted scan cells (prescan design) and you are ready to perform scan assembly. When you use the `insert_dft` command, DFT Compiler stitches the scan registers to form a scan chain, similar to a serial shift register. During scan mode, a combination of patterns are applied to the primary input and shifted out through the scan chain, providing fault coverage for the combinational and sequential logic in the design. For more information, see the DFT Compiler documentation.

Scan Replacement

When you perform a test-ready compile on an unmapped design, the tool maps sequential cells directly to scan registers in your library. If you map a design without using the `-scan` option to the `compile_ultra` command or the `compile` command and then include the `-scan` option in a subsequent compile, the tool uses the scan replacement algorithm as shown in Figure 9-10. However, the recommended method for test-ready compile is to use the `-scan` option of the `compile_ultra` command or the `compile` command on an unmapped design.

Figure 9-10 Mapping to Scan Cells

In the scan replacement flow shown in [Figure 9-11](#), compile first maps sequential cells to nonscan cells from the logic library and then scan-replaces the nonscan cells with scan cells from the logic library.

Figure 9-11 Scan Replacement

Therefore, it is important to have a complete set of nonscan cells as well as equivalent scan cells to obtain the best mapping for test-ready compile. If your library is missing a nonscan equivalent cell for a scan cell, then the tool does not use that scan cell. For example,

assume that you have a scan cell with a synchronous enable pin but you do not have a nonscan cell with this pin. Registers with an enable condition are mapped by using extra logic on the data pin because no nonscan cell is available with an enable pin for the initial mapping.

Identical-function search locates a scan equivalent, using the information in the `test_cell` group of the library description. A valid scan equivalent must meet the following conditions:

- The scan cell and the nonscan cell must have the same functional input and output pins.
- The functional description in the `test_cell` group of the library description must exactly match the functional description of the nonscan cell.

Sequential-mapping search locates a scan equivalent by using the Design Compiler sequential-mapping algorithms. DFT Compiler uses this search only for edge-triggered nonscan sequential cells. Sequential mapping selects the lowest-cost scan equivalent, which can be a scan cell plus combinational gates.

Identical-function search is faster than sequential-mapping mode but might not find a scan equivalent in some cases, such as for complex flip-flops. If neither identical-function search nor sequential-mapping search can find a scan equivalent, DFT Compiler leave the nonscan cell in the design without performing scan replacement and issues the following message:

```
TEST-120 (Warning) No scan equivalent exists for cell
```

If you did not use test-ready compile, DFT Compiler can also perform scan replacement during scan assembly. However, for best results, use test-ready compile to map to scan cells when your starting point is an HDL description.

For additional information about how to assemble scan structures and analyze scan operations, see the DFT Compiler documentation.

Selecting a Scan Style

If you are using test-ready compile to insert scan structures in your design, you must select a scan style before you perform logic synthesis. You must use the same scan style for all modules of your design.

Select a scan style based on your design style and on the types of scan cells available in your target logic library. See the DFT Compiler documentation for more information on considerations for selecting a scan style.

Specify the scan style by setting the `test_default_scan_style` variable. The scan style defined by the `test_default_scan_style` variable applies to all the designs in the current session. You can also use the `set_scan_configuration -style` command to specify the scan style. However, this command applies only to the current design. If your selected scan style differs from the default scan style, you must execute this command for each module.

See the DFT Compiler documentation for details on the `set_scan_configuration` command.

[Table 9-4](#) shows the scan style keywords to use when specifying the scan style. You can use these keywords with either the `test_default_scan_style` variable or the `set_scan_configuration -style` command.

Table 9-4 Scan Style Keywords

Scan style	Keyword
Multiplexed flip-flop	<code>multiplexed_flip_flop</code>
Clocked scan	<code>clocked_scan</code>
Level-sensitive scan design (LSSD)	<code>lssd</code>
Auxiliary-clock LSSD	<code>aux_clock_lssd</code>

Mapping to Libraries Containing Only Scan Registers

Design Compiler automatically maps directly to scan registers from your library without going through a scan-replacement step. Scan-replacement is not possible with a scan-only library. You must always use the `-scan` option of the `compile_ultra` command for these libraries.

Mapping to the Dedicated Scan-Out Pin

Some registers in your library might have both a functional output pin and a dedicated scan-out pin. These dedicated scan-out pins must be identified by the setting following attribute on the output pins in the `.lib` file:

```
test_output_only:true;
```

By default, the `compile -scan` command does not use these pins for functional output connections. This behavior is controlled by the following integer variable (set to 1 by default):

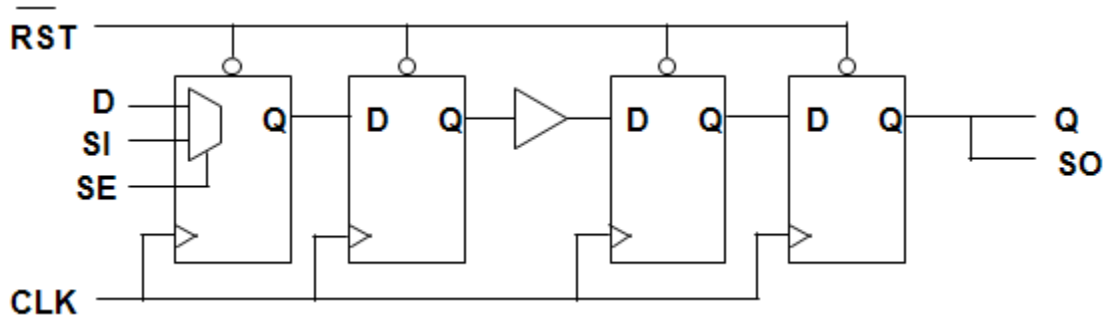
```
set compile_dont_use_dedicated_scanout 1
```

If you observe that `compile` is making use of these pins for functional connections, be sure that this variable is not set to 0 in your scripts, and make sure that your library pins have the correct attribute settings.

Automatic Identification of Shift Registers

Test-ready compile with DC Ultra can automatically identify shift registers in the design and perform scan replacement on only the first register. This capability improves the sequential design area and reduces congestion by using fewer scan-signals for routing. [Figure 9-12](#) shows an example.

Figure 9-12 Example of Shift Register Identification

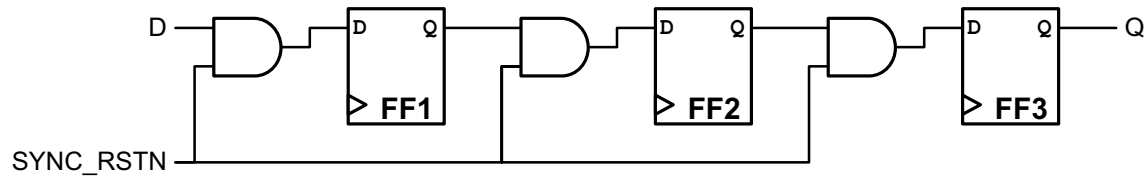


When DC Ultra identifies these shift-registers, DFT Compiler recognizes these identified shift-registers as shift-register scan segments. If required, DFT Compiler breaks these scan segments to satisfy the test setup requirements, such as maximum chain length.

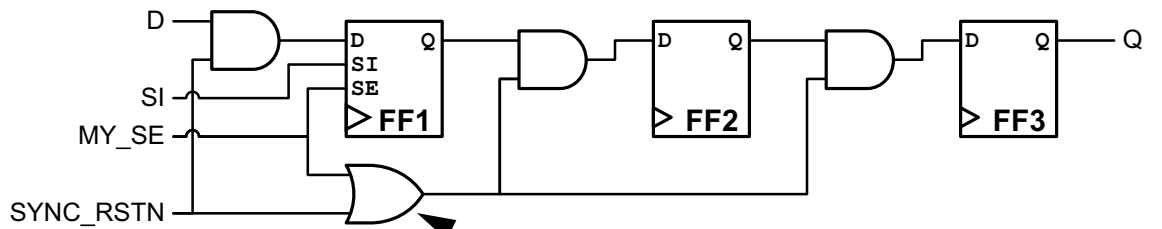
Shift registers containing synchronous logic between the registers can be identified if the synchronous logic is controlled such that the data can be shifted from the output of the first register to the input of the next register. This synchronous logic can either be internal to the register, such as synchronous reset and enable, or it can be external synchronous logic, such as multiplexor logic between the registers. [Figure 9-13](#) shows an example of shift register identification with synchronous reset.

Figure 9-13 Example of Shift Register Identification with Synchronous Reset

After compile_ultra -scan



After insert_dft

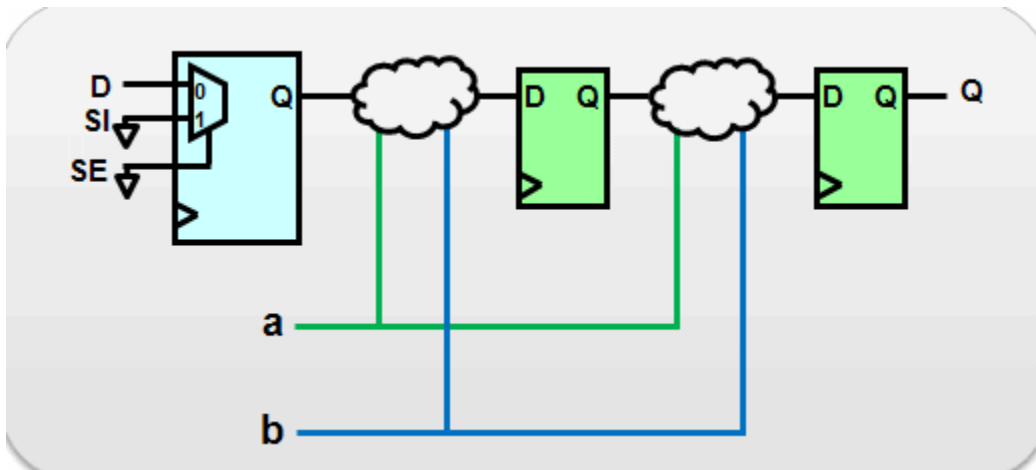


Logic added to scan enable to allow shifting of data

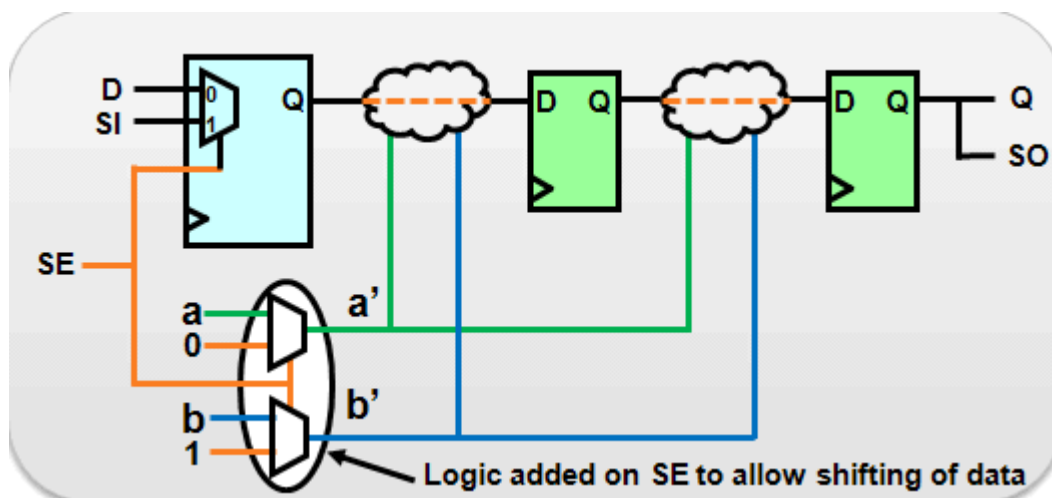
Figure 9-14 shows an example of external synchronous logic with more complex logic between the registers.

Figure 9-14 Example of External Synchronous Logic With Complex Logic Between the Registers

After compile_ultra -scan



After insert_dft



For shift registers identified with synchronous logic between the registers, DFT Compiler adds additional logic to the scan-enable signal during scan insertion. The extra logic allows the data to be shifted between the registers in scan mode. This extra logic results in shared paths between the scan-enable signal and the functional logic. Therefore, not setting the `dont_touch_network` attribute on the scan-enable ports or signals is important. The `dont_touch_network` attribute on the scan-enable signal can propagate into functional logic paths, which prevents optimization of these paths and can lead to QoR degradation.

If the `dont_touch_network` attribute is found on the scan-enable signal in your design and shift registers have been identified where extra logic was inserted on the scan-enable signal, the following warning message is stored in the log file:

```
Warning: dont_touch_network attribute on scan-enable port or signal '%s'
can result in QoR degradation after scan insertion. (TEST-2040).
```

On scan-enable ports, you can use the `set_case_analysis` command to disable timing optimization and the `set_ideal_network` command to disable design rule fixing. However, if you need to use the `dont_touch_network` attribute, use the `dont_touch_network -no_propagate` command instead. By using this command, you can avoid the propagation of the `dont_touch` attribute into the functional logic.

Shift-register identification is available only in DC Ultra with the `compile_ultra -scan` command. When the `compile_seqmap_identify_shift_registers` variable is set to its default, which is `true`, shift register identification is enabled.

The `compile_seqmap_identify_shift_registers_with_synchronous_logic` variable controls the ability to identify shift registers with synchronous logic between the registers. The default is `true`. Set this variable to `false` if you do not want DC Ultra to identify shift registers containing synchronous logic between the registers or if your design flow does not permit the insertion of additional logic on the scan-enable signal.

Note that setting the `compile_seqmap_identify_shift_registers` variable to `false` disables all forms of shift-register identification, including shift registers that contain synchronous logic between the registers.

If you want to exclude specific registers from being identified as shift registers, apply the `dont_use_for_shift_registers` attribute on the registers that should be excluded. For example,

```
dc_shell> set_attribute [get_cells regs_to_exclude] dont_use_for_shift_registers true\
          -type boolean
```

For best results, you should write out the design in `.ddc` format to preserve the information about the shift registers that are identified. When you write out the design in `.ddc` format, the `compile -incremental -scan` command or DFT Compiler can recognize already identified shift registers from a previous compile using the information stored in the `.ddc` binary file.

If your flow requires you to work with an ASCII netlist rather than a `.ddc` file, you must use the `set_scan_state` command when reading the netlist back into Design Compiler. The `set_scan_state` command updates the design to indicate that a test-ready compile has already been performed. The `set_scan_state` command also searches for and re-identifies shift registers in the design.

The following example reads in an ASCII design and uses the `set_scan_state` command to set the scan state status:

```
read_verilog mapped_design.v
current_design top
link
set_scan_state test_ready
```

DC Ultra also supports the identification of shift registers with synchronous logic during the `set_scan_state` command run in the ASCII netlist flow. To enable this capability, you must set the following variable to `true` before you run the `set_scan_state` command:

```
set_app_var compile_seqmap_identify_shift_registers_with_synchronous_logic_ascii true
```

The `compile_seqmap_identify_shift_registers_with_synchronous_logic_ascii` variable is set to `false` by default. When the variable is set to `true`, the `set_scan_state` command performs full identification of all types of shift registers in the design. As a result, the set of newly identified shift registers might be different from the original set of shift registers in the design.

Both of the following default variable settings should be set in all Design Compiler sessions to enable this flow:

```
set_app_var compile_seqmap_identify_shift_registers true
set_app_var compile_seqmap_identify_shift_registers_with_synchronous_logic true
```

When you enable the identification of shift registers with synchronous logic in the ASCII netlist flow, DC Ultra issues the following informational message when running the `set_scan_state` command:

```
Information: Performing full identification of complex shift registers.
(TEST-1190)
```

The following ASCII netlist flow script example includes shift registers with synchronous logic:

```
# Turn on ASCII flow support for shift registers with synchronous logic
set compile_seqmap_identify_shift_registers_with_synchronous_logic_ascii true
read_verilog design.vg
current_design top
link
read_sdc design.sdc
if {[shell_is_in_topographical_mode]} {
extract_physical_constraints floorplan.def
}
# Reapply preexisting DFT constraints (for example, set_scan_element false)
set_scan_state test_ready
# Continue with DFT Compiler flow
```


Using Register Replication to Solve Timing QoR, Congestion, and Fanout Problems

Design Compiler can replicate registers to address timing quality of results (QoR), congestion, and fanout issues. This feature is supported in both Design Compiler topographical mode and wire-load mode. To enable register replication, use the `set_register_replication` command.

For Design Compiler topographical, register replication is placement-aware, which can help reduce congestion in some cases. For wire-load mode, the load of the original replicated register is evenly distributed among the new replicated registers.

To do this	Use this
Specify the value to which the <code>register_replication</code> attribute is to be set (the maximum fanout)	<code>-max_fanout</code>
Specify the value to replicate the registers <code>n</code> times (<code>n >= 2</code>)	<code>-num_copies</code>
Specify a list of registers on which the <code>register_replication</code> attribute is to be set	<code>-object_list</code>

You can either set a `max_fanout` limit on the target register or specify the number of replications by using the `num_copies` option. However, you should check the fanout of the register to be replicated before attempting to use these options. The tool does not replicate registers if the `-max_fanout` or `-num_copies` value is greater than the fanout of the target register.

You can set the `register_replication` attribute only on registers and not on the design or current design. After the register replication feature is enabled, the `compile` command automatically invokes the `set_register_replication` command to replicate registers during optimization. The tool implements the fanout or `num_copies` value that you set by replicating registers but not by inserting buffers.

Keep the following points in mind when you enable the register replication capability:

- If the fanout of the register is seventeen and you specified `-fanout 8` or `-num_copies 3`, then the tool replicates the register three times with fanout 6, 6, and 5. That is, it evenly distributes the fanouts to each register.
- The `compile_ultra -incremental` command supports register replication.

- If you use both the `set_max_fanout` command and the `set_register_replication -max_fanout` command, the `set_register_replication` command has a higher priority.
- If the `-fanout` and `-num_copies` options are applied together, then the `-num_copies` option has the higher priority and the tool prints out the warning message that the `-fanout` option will be ignored.

Example 9-4 shows how to enable register replication on registers.

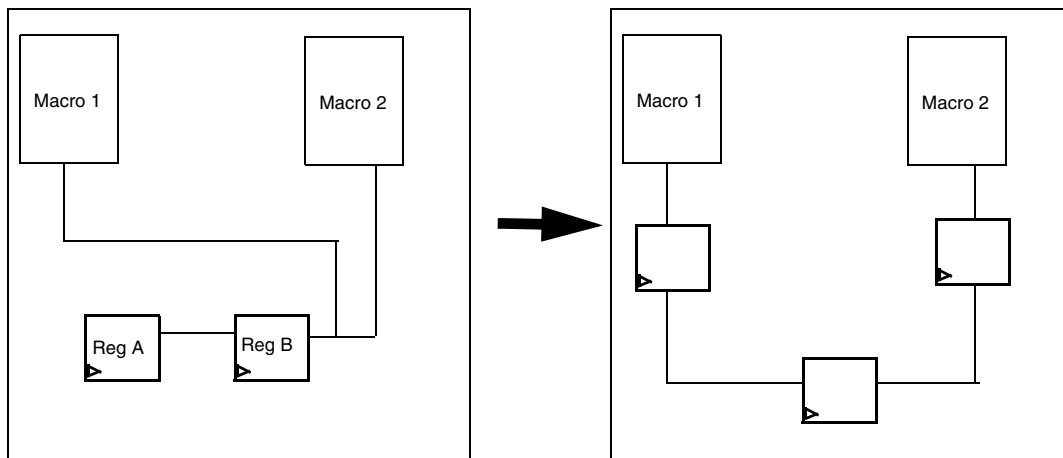
Example 9-4 Enable Register Replication

```
set_register_replication -max_fanout 10 [get_cells -h * -f "@ref_name==*regA*"]
set_register_replication -num_copies 10 [get_cells -h * -f "@ref_name==*regA*"]
```

To specify the style used in naming replicated registers, use the `register_replication_naming_style` variable. The default format is `<%s_rep%d>`. For example, if your original register is named PLL, by default, the replicated registers would be named PLL_rep1, PLL_rep2, PLL_rep3, and so forth.

Figure 9-15 shows a design in which register replication has been enabled. The figure shows a chip that contains two macros. Based on the speed and distance across the logic area of the chip, register B cannot be placed at any location such that timing is met. With the register replication enabled, the tool can replicate register B such that timing is met.

Figure 9-15 Register Replication Example



For complete command syntax, see the man page.

10

Adaptive Retiming

The `compile_ultra` command supports the `-retime` option, which enables Design Compiler to automatically perform register moves during optimization. This capability, called adaptive retiming, enables the tool to move registers and latches to improve timing. Adaptive retiming is intended for use in optimizing general designs; it does not replace the regular retiming engine available with the `optimize_registers` command.

Before you read this chapter, read the [“Optimization Flow” on page 1-9](#) to understand how register retiming fits into the overall compile flow.

Adaptive retiming is supported in both Design Compiler regular mode and topographical mode.

This chapter contains the following sections:

- [Comparing `optimize_registers` With `compile_ultra -retime`](#)
- [Adaptive Retiming Examples](#)
- [Performing Adaptive Retiming](#)
- [Controlling Adaptive Retiming](#)
- [Reporting the `dont_retime` Attribute](#)
- [Removing the `dont_retime` Attribute](#)
- [Verifying Retimed Designs](#)

Comparing `optimize_registers` With `compile_ultra -retime`

Adaptive retiming moves registers and latches to improve worst negative slack (WNS). For datapath designs, you should still use either the `optimize_registers` command or the `set_optimize_registers` command followed by the `compile_ultra` command.

You can use both adaptive retiming and pipelined logic retiming if you use the `set_optimize_registers` command on the pipelined portions of the design prior to running `compile_ultra -retime`, as shown in the following example:

```
set_optimize_registers [get_designs pipelined_ALU]
compile_ultra -retime
```

The commands in the previous example apply pipeline retiming on the pipelined portions of the design and adaptive retiming on the remainder of the design.

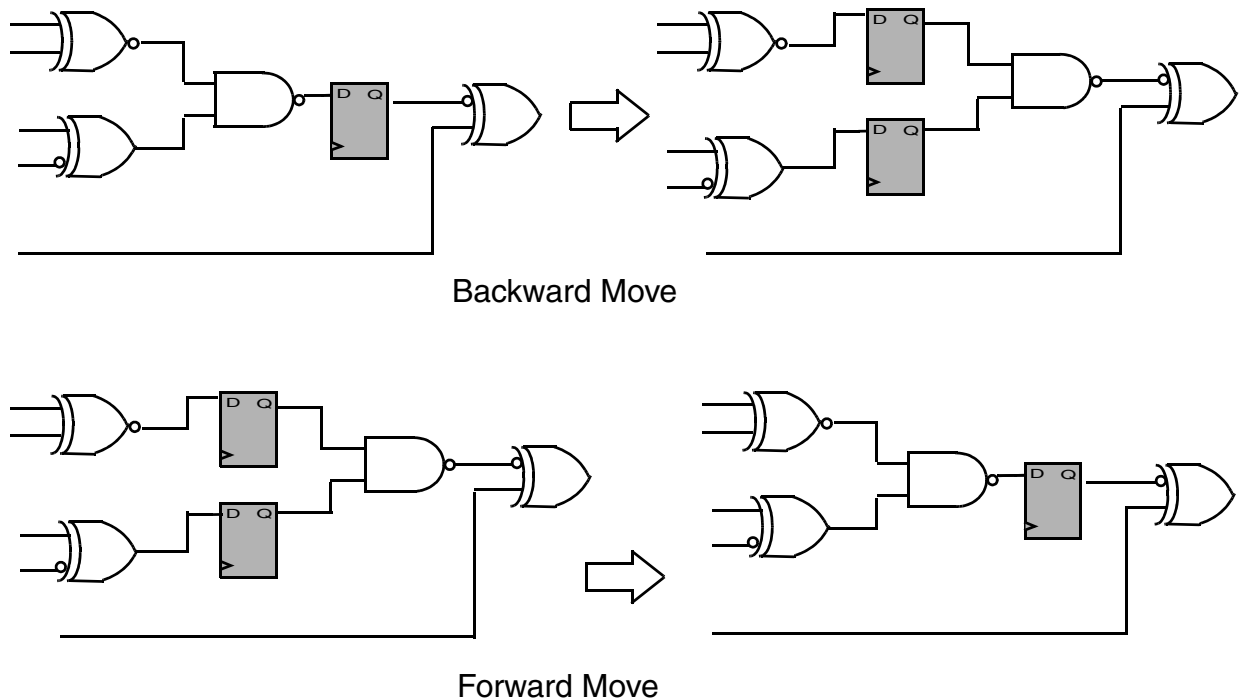
For more information on the pipeline retiming engine, see the *Design Compiler Register Retiming Reference Manual*.

Adaptive Retiming Examples

When you describe circuits prior to logic synthesis, it is usually time-consuming and difficult to find the optimal register locations and code them into the HDL description. With retiming, the locations of the registers and latches in a sequential design can be automatically adjusted to equalize as nearly as possible the delays of the stages. This capability is particularly useful when some stages of a design exceed the timing goal while other stages fall short. If no path exceeds the timing goal, adaptive retiming can be used to reduce the number of registers, where possible.

During retiming, registers are moved forward or backward through the combinational logic of a design as shown in [Figure 10-1](#).

Figure 10-1 Adaptive Retiming



Design Compiler can improve worst negative slack by performing the following tasks during adaptive retiming:

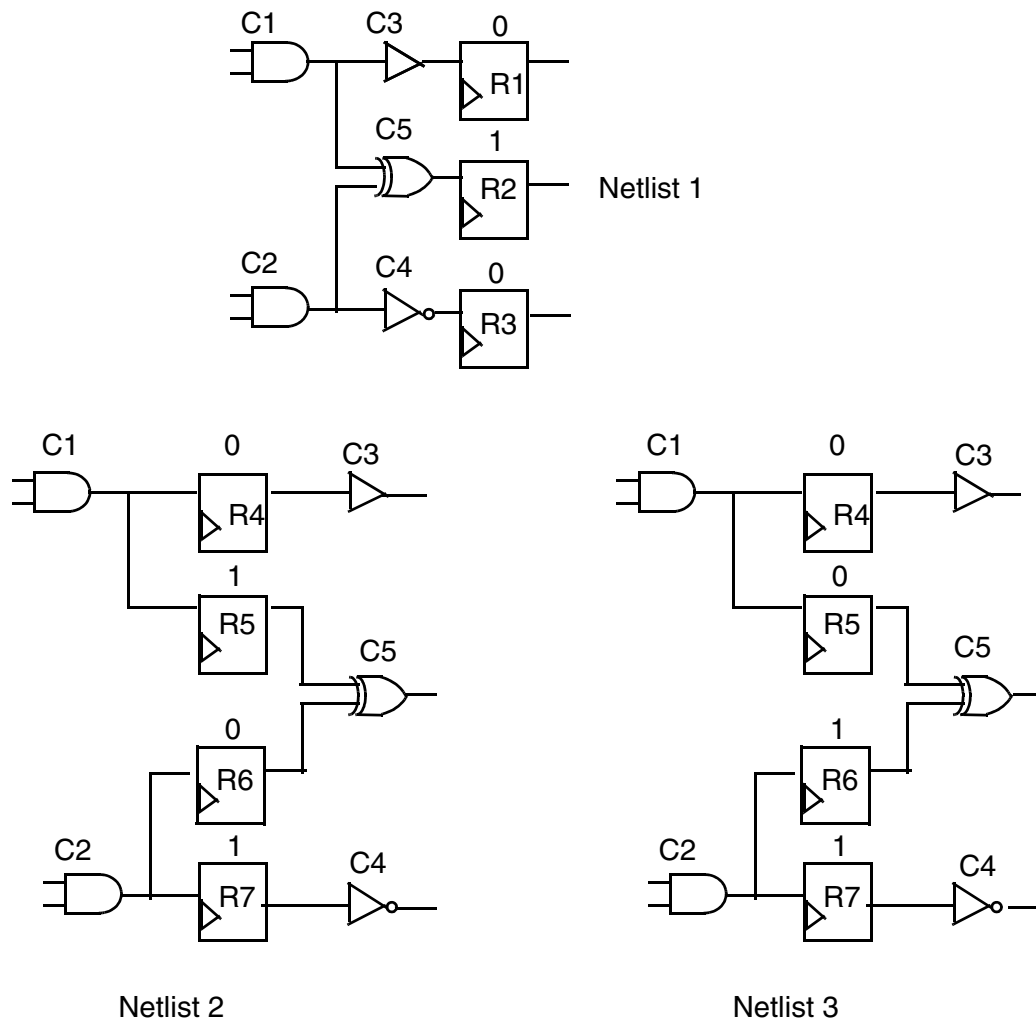
- Retiming registers with conflicting reset requirements
- Retiming registers with extra synchronous input pins (tied high or low)
- Decomposing registers without SEQGEN correspondence

Design Compiler can also improve area by performing the following tasks:

- Allowing forward moves on non-critical registers
- Merging registers with equal and opposite next states

Figure 10-2 shows an example of how Design Compiler handles registers with conflicting reset requirements.

Figure 10-2 Adaptive Retiming for Registers with Conflicting Reset Values

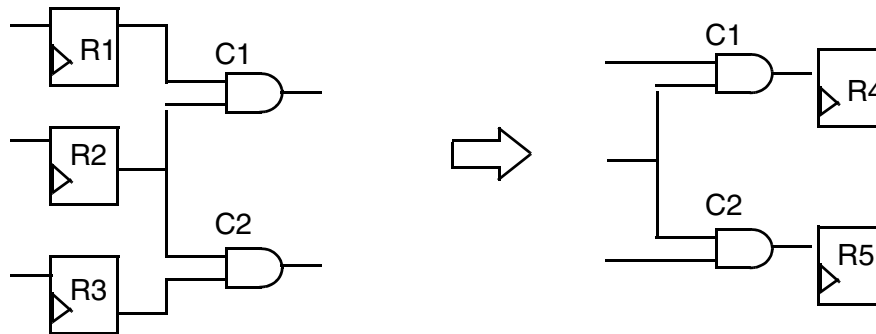


The values (0 or 1) above the registers in the netlists indicate their reset values when a common reset or preset signal (not shown) becomes active. Assume that all three registers R1, R2, R3 in the first netlist move backward across the driver cells of their input nets. The second netlist is a possible result of the moves. Registers R4 and R7 have unique reset or preset values; however, R5 and R6 can have either preset (1) for R5 and reset (0) for R6 as in the second netlist or the reverse as in the third netlist.

Design Compiler can retime registers with these conflicting reset requirements. The two input XOR gate has two possible input assignments resulting in an output value of 1. In the second netlist, conflicting reset values can prevent further backward moves. The third netlist, however, allows such backward moves if they are required.

Figure 10-3 shows how Design Compiler can execute forward local retiming moves on non-critical registers to improve area by decreasing the number of registers.

Figure 10-3 Forward Moves for Non-Critical Registers



Design Compiler can move registers with common timing exceptions, including `max_delay`, `min_delay`, `multicycle_path`, `false_path`, and `group_path`, as long as all the registers being moved have the same exceptions. After executing a move involving registers with exceptions, the new registers inherit all the exceptions from the original registers.

Two registers have the same exceptions if they appear in the same point-to-point timing exception setting commands, such as `set_max_delay`, `set_min_delay`, `set_multicycle_path`, `set_false_path`, and `group_path`.

In the following example, registers `r0` and `r1` have the same exceptions.

```
set_max_delay 10 -to [list r0/D r1/D]
```

Alternatively, if this `set_max_delay` command is split into two `set_max_delay` commands, both with the same delay value, the two registers still have the same exceptions:

```
set_max_delay 10 -to r0/D
set_max_delay 10 -to r1/D
```

However, if the two `set_max_delay` commands have different delay values, the two registers have different exceptions, and they cannot be moved together:

```
set_max_delay 10 -to r0/D
set_max_delay 15 -to r1/D
```

By default this feature is disabled: to enable this feature, set the `compile_retime_exception_registers` variable to true.

Performing Adaptive Retiming

You use the `-retime` option of the `compile_ultra` command to perform adaptive retiming and to improve the delay during optimization.

Adaptive retiming honors attributes, such as `dont_touch`, `size_only`, and `dont_retime`; retiming is prevented if these attributes are set. Registers that are moved as a result of adaptive retiming are renamed with a prefix of R and a numbered suffix (R_xxx); you cannot match these retimed registers to the original registers.

Adaptive retiming supports all `compile_ultra` options except the following:

- `-top`
- `-only_design_rule`

Controlling Adaptive Retiming

You can use the `set_dont_retime` command to include or exclude designs or cells from being retimed. For example, the following command specifies that the design a1 should not be retimed:

```
set_dont_retime [get_designs a1] true
```

Setting the `dont_retime` attribute on a hierarchical cell implies that the attribute is also set on all cells below it. However, you can override the attribute at a lower-level of the hierarchy by explicitly setting the `dont_retime` attribute at that level. For example, the following command specifies that the cells `z1_reg` and `z2_reg` should be retimed:

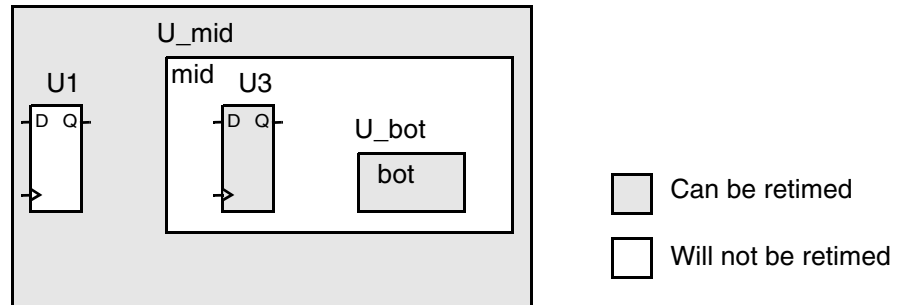
```
set_dont_retime [get_cells {z1_reg z2_reg}] false
```

If you use the `set_dont_retime` command without specifying either true or false, a value of true is assumed.

The following sequence of commands will create the scenario shown in [Figure 10-4](#):

```
set_dont_retime [get_cells U1]
set_dont_retime [get_designs mid] true
set_dont_retime [get_cells U_mid/U3] false
set_dont_retime [get_cells U_mid/U_bot] false
compile_ultra -retime
```


Figure 10-4 Design With Both Retimable and Nonretimable Cells

**Note:**

The `dont_retime` attribute on a child cell has priority over the attribute set on any ancestor cell or design.

You can also use the `set_dont_retime` command to prevent retiming on specific cells or designs when you use the `optimize_registers` command. For more information, see the *Design Compiler Register Retiming Reference Manual*.

Reporting the dont_retime Attribute

Use the `report_attribute` or `get_attribute` command to check which cells or designs have the `dont_retime` attribute.

Removing the dont_retime Attribute

To remove the `dont_retime` attribute, use the `remove_attribute` command.

Verifying Retimed Designs

When Design Compiler performs any retiming, an automated setup file is necessary for verification with Formality. The automated setup file includes retiming optimization information that helps Formality to verify retimed designs. To remind you that Formality requires an automated setup file, Design Compiler displays the following message when retiming is performed:

```
Information: Retiming is enabled. SVF file must be used for
formal verification. (OPT-1210)
```

If you run multiple `compile_ultra -retime` commands, you must run multiple formal verifications, each one verifying only the changes made by the most recent `compile_ultra -retime` command.

The following example shows a Design Compiler script that runs the `compile_ultra -retime` command followed by an incremental compile with the `-retime` option. In the script, the verification is run by comparing the RTL to the `compile_1.ddc` netlist using the `compile_1.svf` verification setup file and comparing the `compile_1.ddc` netlist to the `compile_2.ddc` netlist using the `compile_2.svf` verification setup file.

```
set_svf compile_1.svf
read_verilog rtl.v
compile_ultra -retime
write -hierarchy -output compile_1.ddc
set_svf compile_2.svf
compile_ultra -incremental -retime
write -hierarchy -output compile_2.ddc
```

11

Gate-Level Optimization

During gate-level optimizations, Design Compiler implements the final netlist by making optimal selections of library cells. It performs tasks such as delay optimization, design rule fixing, and area recovery. Design Compiler's optimization algorithms use costs to determine if a design change is an improvement. Design Compiler calculates two cost functions: one for design rule constraints and one for optimization constraints and accepts an optimization move if it decreases the cost of one component without increasing more-important costs. Before you read this chapter, read the [“Optimization Flow” on page 1-9](#) to understand how gate-level optimizations fit into the overall compile flow.

This chapter contains the following sections:

- [Compile Cost Function](#)
- [Changing the Cost Function](#)
- [Compile Log](#)
- [Delay Optimization](#)
- [Design Rule Fixing](#)
- [Area Recovery](#)

Compile Cost Function

During gate-level optimization, Design Compiler calculates two cost functions: one for design rule constraints and one for optimization constraints. Cost calculations are affected by constraints such as user-specified constraints (for example, `create_clock`), library constraints (for example, `max_fanout`), and built-in optimization goals (for example, area cleanup). A cost function consists of deltas from these constraints, that is, the positive difference between the actual value and target value of the constraint. When evaluating cost function components, Design Compiler considers these violators and works to reduce the cost function to zero.

Design Compiler reports the value of each cost function whenever a change is made to the design. The compile cost function considers only those components that are active in your design. Design Compiler evaluates each cost function component independently, in order of importance.

Design Compiler evaluates cost function components independently in order of importance and accepts an optimization move if it decreases the cost of one component without increasing more-important costs. For example, an optimization move that improves maximum delay cost is always accepted. Optimization stops when all costs are zero or no further improvements can be made to the cost function.

Design Rules Cost Function

Design rule constraints reflect technology-specific restrictions your design must meet in order to function as intended. The design rules cost function has the components shown in [Figure 11-1](#).

Figure 11-1 Design Rules Cost Function

$$\text{Cost} = \sum \Delta \text{max_fanout} + \sum \Delta \text{max_transition} + \sum \Delta \text{max_capacitance}$$

The design rules cost function takes into account the following design rule constraints:

- Maximum transition time
- Maximum fanout
- Maximum capacitance

Calculating Transition Time Cost

The maximum transition time for a net is the longest time required for its driving pin to change logic values. Design Compiler determines driver transition times from the logic library. If the transition time for a given driver is greater than the `max_transition` value, Design Compiler reports a design rule violation and works to correct the violation.

Calculating Fanout Cost

Most technology libraries place fanout restrictions on driving pins, creating an implicit fanout constraint for every driving pin in designs using that library. Design Compiler computes fanout load for a driver by using the following equation:

$$\sum_{i=1}^m fanout_load_i$$

m is the number of inputs driven by the driver.

$fanout_load_i$ is the fanout load of the i th input.

If the calculated fanout load is greater than the `max_fanout` value, Design Compiler reports a design rule violation and attempts to correct the violation.

Calculating Capacitance Cost

The maximum capacitance is a pin-level attribute used to define the maximum total capacitive load that an output pin can drive. Design Compiler computes the total capacitance for a driver by using the following equation:

$$\sum_{i=1}^m C_i$$

m is the number of inputs driven by the driver.

C_i is the capacitance of the i th input.

If the calculated capacitance is greater than the `max_capacitance` value, Design Compiler reports a design rule violation and attempts to correct the violation.

Optimization Constraints Cost Function

Optimization constraints represent speed and area design goals and restrictions; speed (timing) constraints have higher priority than area. The optimization constraints cost function has the components shown in [Figure 11-2](#).

Figure 11-2 Optimization Constraints Cost Function

$$\text{Cost} = \Sigma \Delta \text{ max_delay} + \Sigma \Delta \text{ min_delay} + \Sigma \Delta \text{ max_area}$$

The full optimization cost function takes into account the following components, listed in order of importance. Not all components are active on all designs.

- Maximum delay cost
- Minimum delay cost
- Maximum area cost

Calculating Maximum Delay Cost

Maximum delay is usually the most important portion of the optimization cost function. Maximum delay target values for each timing path in the design are automatically determined after considering clock waveforms and skew, library setup times, external delays, multicycle or false path specifications, and the `set_max_delay` commands. Load, drive, operating conditions, wire load model, and other factors are also taken into account.

Design Compiler supports two methods for calculating the maximum delay cost:

- Worst negative slack (default behavior)
- Critical range negative slack

The following sections describe these methods.

Worst Negative Slack Method

By default, Design Compiler uses the worst negative slack method to calculate the maximum delay cost. With the worst negative slack method, only the worst violator in each path group is considered.

A path group is a collection of paths that to Design Compiler represent a group in maximum delay cost calculations. Each time you create a clock with the `create_clock` command, Design Compiler creates a path group that contains all the paths associated with the clock. You can also create path groups by using the `group_path` command. Design Compiler places in the default group any paths that are not associated with any particular group or

clock. To see the path groups defined for your design, run the `report_path_group` command.

Because the worst negative slack method does not optimize near-critical paths, this method requires fewer CPU resources than the critical negative slack method. Because of the shorter runtimes, the worst negative slack method is ideal for the exploration phase of the design. Always use the worst negative slack method during default compile runs.

With the worst negative slack method, the equation for the maximum delay cost is

$$\sum_{i=1}^m v_i \times w_i$$

m is the number of path groups.

v_i is the worst violator in the i th path group.

w_i is the weight assigned to the i th path group (the default is 1.0).

Design Compiler calculates the maximum delay violation for each path group as

`max (0, (actual_path_delay - max_delay))`

Because only the worst violator in each path group contributes to the maximum delay violation, how you group paths affects the maximum delay cost calculation.

- If only one path group exists, the maximum delay cost is the amount of the worst violation multiplied by the group weight.
- When multiple path groups exist, the costs for all the groups are added to determine the maximum delay cost of the design.

During optimization, the Design Compiler focus is on reducing the delay of the most critical path. This path changes during optimization. If Design Compiler minimizes the initial path's delay so that it is no longer the worst violator, the tool shifts its focus to the path that is now the most critical path in the group.

Critical Range Negative Slack Method

Design Compiler also supports the critical range negative slack method to calculate the maximum delay cost. The critical range negative slack method considers all violators in each path group that are within a specified delay margin (referred to as the critical range) of the worst violator.

For example, if the critical range is 2.0 ns and the worst violator has a delay of 10.0 ns, Design Compiler optimizes all paths that have a delay between 8.0 and 10.0 ns.

The critical range negative slack is the sum of all negative slack values within the critical range for each path group. When the critical range is large enough to include all violators, the critical negative slack is equal to the total negative slack.

Using the critical negative slack method, the equation for the maximum delay cost is

$$\sum_{i=1}^m \left(\sum_{j=1}^n v_{ij} \right) \times w_i$$

m is the number of path groups.

n is the number of paths in the critical range in the path group.

v_{ij} is a violator within the critical range of the i th path group.

w_i is the weight assigned to the i th path group.

Design Compiler calculates the maximum delay violation for each path within the critical range as

$$\max (0, (\text{actual_path_delay} - \text{max_delay}))$$

Calculating Minimum Delay Cost

The equation for the minimum delay cost is

$$\sum_{i=1}^m v_i$$

m is the number of paths affected by `set_min_delay` or `set_fix_hold`.

v_i is the i th minimum delay violation.

Design Compiler calculates the minimum delay violation for each path as

$$\max (0, (\text{min_delay} - \text{actual_path_delay}))$$

The minimum delay cost for a design differs from the maximum delay cost. Path groups do not affect the minimum delay cost. In addition, all violators, not just the most critical path, contribute to the minimum delay cost.

Calculating Maximum Power Cost

Design Compiler computes the maximum power cost only if you have a Power-Optimization license and your logic library is characterized for power.

The maximum power cost has two components:

- Maximum dynamic power

Design Compiler calculates the maximum dynamic power cost as

```
max (0, actual_power - max_dynamic_power)
```

- Maximum leakage power

Design Compiler calculates the maximum leakage power cost as

```
max (0, actual_power - max_leakage_power)
```

For more information about the maximum power cost, see the *Power Compiler User Guide*.

Calculating Maximum Area Cost

Design Compiler computes the area of a design by summing the areas of each of its components (cells) on the design hierarchy's lowest level (and the area of the nets). Design Compiler ignores the following components when calculating circuit area:

- Unknown components
- Components with unknown areas
- Technology-independent generic cells

The cell and net areas are technology dependent. Design Compiler obtains this information from the logic library.

Design Compiler calculates the maximum area cost as

```
max (0, actual_area - max_area)
```

Changing the Cost Function

You can change the cost function described in [“Compile Cost Function” on page 11-2](#) in the following ways:

- Reorder priorities of constraints by using the `set_cost_priority` command
- Disable evaluation of the design rule cost function or optimization constraints cost function by using the `-no_design_rule` or `-only_design_rule` option of the `compile_ultra` or `compile` command
- Prioritize area over total negative slack by using the `set_max_area` command

Reordering the Default Priority of Constraints

Design Compiler tries to meet both design rule constraints and optimization constraints, but design rule constraints take precedence. [Table 11-1](#) shows the default order of priorities. The table shows that, by default, design rule constraints have priority over optimization constraints. However, you can reorder the priorities of the constraints listed in **bold** type by using the `set_cost_priority` command.

Table 11-1 Constraints Default Cost Vector

Priority (descending order)	Notes
connection classes	
multiple_port_net_cost	
min_capacitance	Design rule constraint
max_transition	Design rule constraint
max_fanout	Design rule constraint
max_capacitance	Design rule constraint
cell_degradation	Design rule constraint
max_delay	Optimization constraint
min_delay	Optimization constraint
power	Optimization constraint

Table 11-1 Constraints Default Cost Vector (Continued)

Priority (descending order)	Notes
area	Optimization constraint
cell count	

The following are circumstances under which you might want to move the optimization constraint `max_delay` ahead of the maximum design rule constraints.

- In many technology libraries, the only significant design rule violations that cannot be fixed without hurting delay are overconstrained nets, such as input ports with large external loads or around logic marked `dont_touch`. Placing `max_delay` ahead of the design rule constraints in priority allows these design rule constraint violations to be fixed in a way that does not hurt delay. Design Compiler might, for example, resize the drivers in another module.
- In compilation of a small block of logic, such as an extracted critical region of a larger design, the possibility of overconstraints at the block boundaries is high. In this case, design rule fixing might better be postponed until the small block has been regrouped into the larger design.

The syntax is

```
set_cost_priority [-default] [-delay] cost_list
```

`-default`

Directs Design Compiler to use its default priority, as shown in [Table 11-1 on page 11-8](#).

`-delay`

Specifies that `max_delay` has higher priority than the maximum design rule constraints.

`cost_list`

Specifies the order of priority (listing the highest first) of the following costs: `max_delay`, `min_delay`, `max_transition`, `max_fanout`, `max_capacitance`, `cell_degradation`, and `max_design_rules`.

Note:

Use of the `cost_list` option requires a DC Ultra license.

Disabling the Cost Function

You can direct Design Compiler to avoid design rule fixing or to compile with only design rule fixing. Use the following options to the `compile_ultra` command or `compile` command to control design rule fixing:

- The `-no_design_rule` option causes compile to exit before fixing design rule violations. This allows you to check the results in a constraint report.
- The `-only_design_rule` option causes compile to perform only design rule fixing.
- The `-only_hold_time` option causes compile to fix only hold time. Design Compiler fixes hold time requirements only when directed by the `set_fix_hold` command. The `compile_ultra` command does not support this option.

Prioritizing Area Over Total Negative Slack

Use the `-ignore_tns` option of the `set_max_area` command to prioritize area over total negative slack during area optimization. When you use this option, the `ignore_tns` attribute is set on the design and compile might increase delay violations at an endpoint in order to improve area (as long as the new delay violation is smaller than the violation on the endpoint of the most critical path in the same path group).

Compile Log

The compile log records the status of the compile run. Each optimization task has an introductory heading, followed by the actions taken while that task is performed. There are three tasks in which Design Compiler works to reduce the compile cost function described in [“Compile Cost Function” on page 11-2](#):

- Delay optimization (see [“Delay Optimization” on page 11-12](#))
- Design rule fixing (see [“Design Rule Fixing” on page 11-14](#))
- Area recovery (see [“Area Recovery” on page 11-15](#))

While completing these tasks, Design Compiler performs many trials to determine how to reduce the cost function. For this reason, these tasks are collectively known as the trials phase of optimization. The compile log displays reduction in costs as shown in [Example 11-1](#). You can customize the trials phase output by setting the `compile_log_format` variable.

Example 11-1 Compile Log

Beginning Delay Optimization Phase

ELAPSED TIME	AREA	WORST NEG SLACK	TOTAL NEG SLACK	DESIGN RULE COST	ENDPOINT
0:00:18	15003.8	0.00	0.0	33.8	
0:00:18	15003.8	0.00	0.0	33.8	
0:00:18	15003.8	0.00	0.0	33.8	
0:00:18	15003.8	0.00	0.0	33.8	
0:00:18	15003.8	0.00	0.0	33.8	
0:00:18	15003.8	0.00	0.0	33.8	
0:00:18	15003.8	0.00	0.0	33.8	

Beginning Design Rule Fixing (max_capacitance)

ELAPSED TIME	AREA	WORST NEG SLACK	TOTAL NEG SLACK	DESIGN RULE COST	ENDPOINT
0:00:18	15003.8	0.00	0.0	33.8	
0:00:19	15172.8	0.00	0.0	0.0	
0:00:20	15172.8	0.00	0.0	0.0	
0:00:20	15172.8	0.00	0.0	0.0	
0:00:20	15172.8	0.00	0.0	0.0	

Beginning Area-Recovery Phase (max_area 0)

ELAPSED TIME	AREA	WORST NEG SLACK	TOTAL NEG SLACK	DESIGN RULE COST	ENDPOINT
0:00:20	15172.8	0.00	0.0	0.0	
0:00:21	15085.7	0.00	0.0	0.0	
0:00:21	15085.7	0.00	0.0	0.0	
0:00:21	15085.7	0.00	0.0	0.0	
0:00:21	15085.7	0.00	0.0	0.0	

Loading db file '/remote/srm147/LS_IMAGES/D20061217/libraries/syn/
tc6a_cbacore.db'

Optimization Complete

1

For more information about log files, see the chapter on analyzing and resolving design problems in the *Design Compiler User Guide*.

Delay Optimization

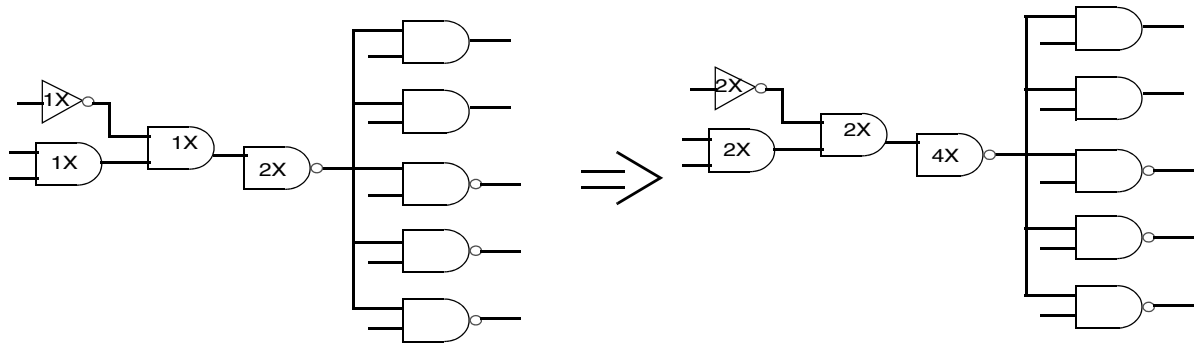
In this phase, Design Compiler attempts to fix existing delay violations by traversing the critical path. It applies local transformations such as upsizing, load isolation and splitting, and revisits mapping of sequential paths.

[Figure 11-3](#) shows some common local optimization steps. Design Compiler takes design rules into account during this phase. When two circuit solutions offer the same delay performance, Design Compiler implements the solution that has the lower design rule cost.

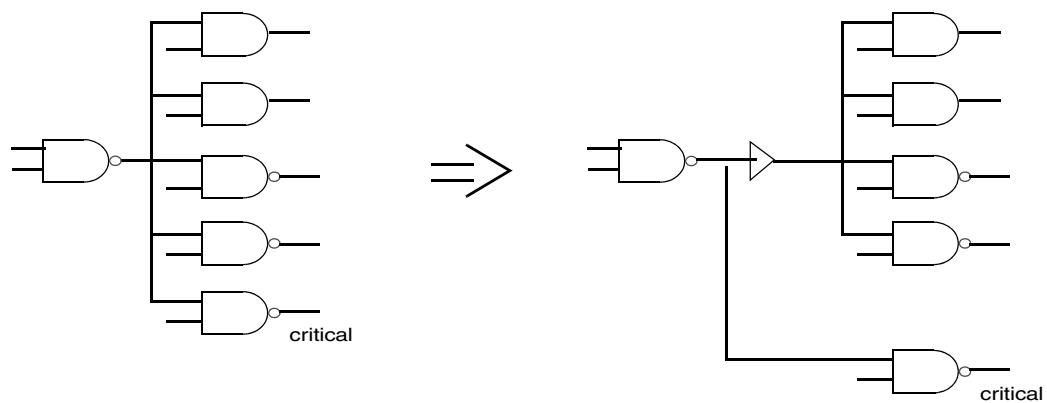
Design Compiler also performs critical path resynthesis on your design improves timing. It identifies the critical path and attempts to do a full compile on only the logic along that path. This process then repeats on the new critical path. The `compile` command, with its `-map_effort high`, option enables critical path resynthesis.

Additionally, Design Compiler resizes or speeds up sequential cells on the critical path. The tool also uses a high-effort algorithm to remove sequential elements for which the logic leading to a constant value is particularly complex. If you do not want Design Compiler to perform this extra optimization, set the `compile_seqmap_propagate_high_effort` variable to false (the default is true). For more information, see [“Automatically Removing Unnecessary Registers” on page 9-9](#).

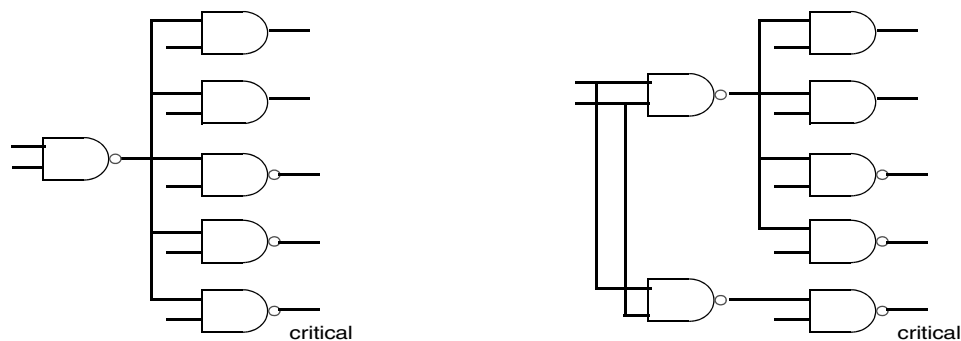
Figure 11-3 Delay Optimization Steps
Upsizing



Load Isolation



Load Splitting



The compile log displays the delay optimization phase as shown in [Example 11-2](#).

Example 11-2 Delay Optimization in Compile Log

Beginning Delay Optimization Phase

ELAPSED TIME	AREA	WORST NEG SLACK	TOTAL NEG SLACK	DESIGN RULE COST	ENDPOINT
0:00:05	136	2.11	23.2	18.0	out_reg[10]/D
0:00:05	138	1.53	16.9	18.0	out_reg[10]/D

Design Rule Fixing

During this phase, the goal is to correct design rule violations by inserting buffers or resizing existing cells. Design Compiler tries to fix these violations without affecting timing and area results, but if necessary, it does violate the optimization constraints. Design rules are provided in the vendor logic library to ensure that the product meets specifications and works as intended. Whenever possible, Design Compiler fixes design rule violations by resizing gates across multiple logic levels—as opposed to adding buffers to the circuitry.

The compile log displays the design rule fixing phase as shown in [Example 11-3](#).

Example 11-3 Design Rule Fixing in Compile Log

Beginning Design Rule Fixing

(max_capacitance) (max_fanout) (max_capacitance)

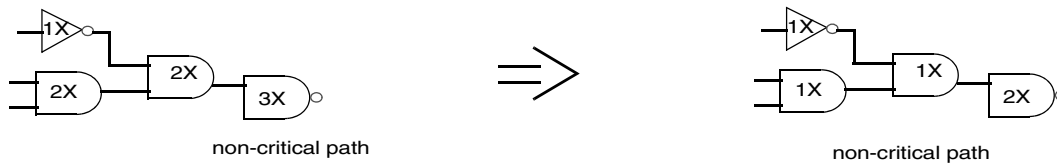
ELAPSED TIME	AREA	WORST NEG SLACK	TOTAL NEG SLACK	DESIGN RULE COST	ENDPOINT
0:00:08	153	0.60	6.6	8.0	
0:00:08	146	0.60	6.6	5.0	

You can direct Design Compiler to avoid design rule fixing or to compile with only design rule fixing. See [“Changing the Cost Function” on page 11-8](#).

Area Recovery

If you have placed area constraints on your design (with the `set_max_area` command), Design Compiler now attempts to minimize the number of gates in the design. The goal is to reduce design area without introducing design rule and delay violations. The tool downsizes cells to recover area. It performs downsizing only on paths that have positive slack as shown in [Figure 11-4](#).

Figure 11-4 Example of Downsizing Cells During Area Recovery



In addition, the tool includes a fast back-end sequential mapper that does automatic sequential area recovery. It identifies clusters of registers with similar functionality and timing and optimizes the area of these register clusters as a whole. See [“Automatically Removing Unnecessary Registers” on page 9-9](#).

The compile log displays the area recovery phase as shown in [Example 11-4](#).

Example 11-4 Area Recovery in Compile Log

```
Beginning Area-Recovery Phase (max_area 145)
```

ELAPSED TIME	AREA	WORST NEG SLACK	TOTAL NEG SLACK	DESIGN RULE COST	ENDPOINT
0:00:09	149.0	1.05	8.5	0.0	

If you do not place area constraints on your design, Design Compiler performs a limited series of downsizing and area cleanup steps as shown in [Example 11-5](#).

Example 11-5 Area Recovery in Compile Log

```
Beginning Area-Recovery Phase (cleanup)
```

ELAPSED TIME	AREA	WORST NEG SLACK	TOTAL NEG SLACK	DESIGN RULE COST	ENDPOINT
0:00:10	16413.4	0.00	0.0	0.0	
0:00:10	16413.4	0.00	0.0	0.0	
0:00:10	16408.2	0.00	0.0	0.0	

Using the `-map_effort` or `-area_effort` option of the `compile` command, you can direct Design Compiler to put a medium, or high effort into area optimization.

- **Medium effort**
Design Compiler does gate sizing and buffer and inverter cleanup.
- **High effort**
Design Compiler tries still more gate minimization strategies. The tool adds gate composition to the process and allocates more CPU time than medium effort.

Note:

Whichever area optimization effort level you choose, the overall constraints cost vector (described in [“Compile Cost Function” on page 11-2](#)) prevails. Even during area optimization, if Design Compiler finds a new opportunity to improve delay cost, it makes the change—even if it increases area cost. Area always has a lower priority than delay.

12

Incremental ASCII Flow With a Third-Party DFT Flow Example

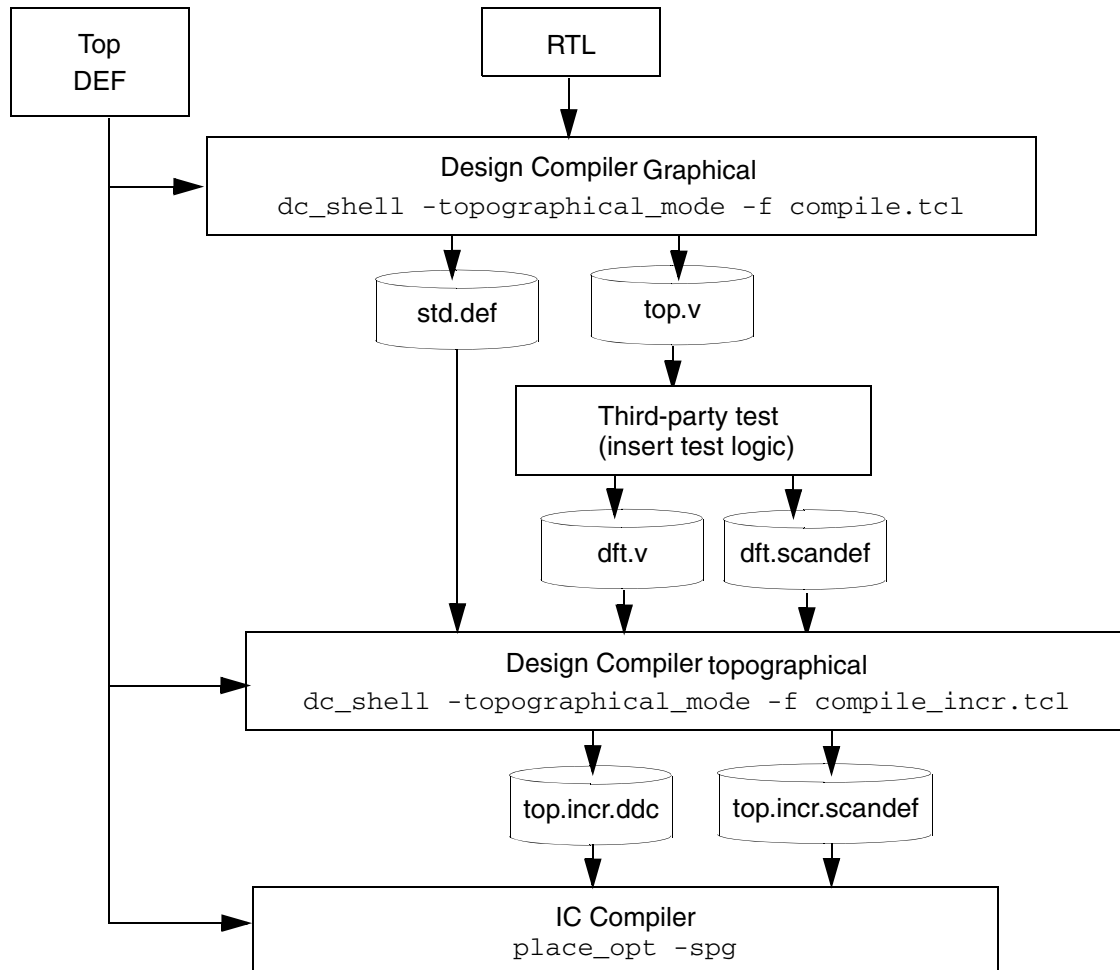
Some flows require you to use a Verilog or VHDL (ASCII) netlist output format. This chapter describes the incremental ASCII flow using a third-party design-for-test (DFT) flow as an example. The flow is described in `compile.tcl` and `compile_incr.tcl` scripts that are typical Tcl scripts that you can use for the initial compilation and incremental compilation flows.

This chapter includes the following sections:

- [Performing Initial Compilation Using the `compile.tcl` Script](#)
- [Performing Incremental Compilation Using the `compile_incr.tcl` Script](#)
- [Using the Netlist SCANDEF Flow](#)

Figure 12-1 shows an example of an incremental ASCII physical guidance flow with a third-party DFT flow.

Figure 12-1 Incremental ASCII Physical Guidance Flow With a Third-Party DFT Flow



Performing Initial Compilation Using the compile.tcl Script

[Example 12-1](#) shows a compile.tcl script that you can use for a typical initial compile_ultra physical guidance flow. It uses Design Compiler Graphical floorplan exploration after the compile_ultra step in order to generate the complete floorplan constraints that contain the standard cell placement. The floorplan exploration step is run with the start_icc_dp -f icc_dp.tcl command.

Example 12-1 Initial Compilation Using the compile.tcl Script

```
# Analyze and elaborate step
analyze -f verilog top_design.v
elaborate top_design

# Load UPF constraints
load_upf upf_constraints.upf

# Set timing constraints
source timing_constraints.tcl

# Set power settings if required
# set_clock_gating_style
# set_max_leakage_power
# set_max_dynamic_power

extract_physical_constraints -allow_physical_cells top.def

compile_ultra -spg -scan -gate_clock

change_names -rules verilog -hierarchy
set write_sdc_output_net_resistance false
set write_sdc_output_lumped_net_capacitance false
write_sdc top.sdc
save_upf top.upf
write_file -f verilog -hierarchy -output top.v
start_icc_dp -f icc_dp.tcl
```

The icc_dp.tcl script generates the standard cell placement so you can reuse it during the compile_ultra -incremental step. [Example 12-2](#) shows the commands included in the icc_dp.tcl script:

Example 12-2 Commands in icc_dp.tcl Script

```
write_def -version 5.7 -component -verbose -output std.def
exit
```

Performing Incremental Compilation Using the `compile_incr.tcl` Script

[Example 12-3](#) shows a `compile_incr.tcl` script that you can use for a typical incremental `compile_ultra` physical guidance flow. The script reads in the Verilog netlist after DFT insertion and reads the placement from the first `compile_ultra` run so that the incremental step can use the standard cell placement as seed placement during placement optimization.

If your initial compilation used physical guidance, as is the case in [Example 12-3](#), you must specify the `-standard_cell spg` option with the `extract_physical_constraints` command to read the standard cell placement in addition to the floorplan constraints. When you run the incremental compilation, specify the `-spg` option.

If your initial compilation did not use physical guidance, use the `-standard_cell topo` option with the `extract_physical_constraints` command to read the standard cell placement in addition to the floorplan constraints. If you do not use the `-standard_cell topo` option, Design Compiler is forced to redo a full placement. This can hurt runtime and correlation because the incremental step cannot use the standard cell placement from the original `compile_ultra` run.

The `compile_incr.tcl` script also reads the SCANDEF file from the DFT insertion step using the `read_scan_def` command and runs the `write_scan_def` command after the `compile_ultra -incremental` command in order to pass the SCANDEF file with all the changes performed during the incremental optimization to IC Compiler. You must use the `-output` option with the `write_scan_def` command if the handoff to IC Compiler is in ASCII format.

For more information about using the netlist SCANDEF flow to import and export scan chain information when the test implementation has been performed outside the Synopsys flow, see [“Using the Netlist SCANDEF Flow” on page 12-5](#).

After you run the `compile_ultra -incremental` command step, you can save the design in .ddc format, along with the SCANDEF file, and proceed with physical implementation in IC Compiler with the `place_opt -optimize_dft` command.

Example 12-3 Incremental Compilation Using the `compile_incr.tcl` Script

```
# Read the DFT netlist
read_verilog top.dft.v

# Load UPF constraints
load_upf top.upf

# Set timing constraints
read_sdc top.sdc

# Restore floorplan constraints and standard cell placement from the
# compile_ultra step
extract_physical_constraints -allow_physical_cells top.def
```

```
extract_physical_constraints -allow_physical_cells \  
                           -standard_cell spg std.def  
  
# Read the SCANDEF file from the DFT insertion step  
set dct_placement_ignore_scan true  
read_scan_def dft.scandef  
  
compile_ultra -incremental -spg  
  
write_scan_def -output top.incr.scandef  
write_file -f ddc -hierarchy -output top.incr.ddc
```

Using the Netlist SCANDEF Flow

Design Compiler can import and export scan chain information when the test implementation has been performed outside the Synopsys flow with the netlist SCANDEF flow. In the netlist SCANDEF flow, you use the `read_scan_def` and `write_scan_def` commands to identify and preserve the scan chain structure and write out the new scan chain structure after optimization.

The scan chains are described in a SCANDEF file. The SCANDEF format, which is part of the DEF open format, is used to describe the scan chain elements. You use the `read_scan_def` command to read the SCANDEF file, after reading the netlist, to identify all cells that are part of scan chains. This identification allows the `write_scan_def` command to generate the new SCANDEF file with the same scan chain structure after optimization. You can then pass the SCANDEF file with all the changes performed during the incremental optimization, such as ungrouping, to IC Compiler.

Design Compiler only preserves the scan chains. It does not reorder them. You cannot change or add new scan chains. Therefore, the `insert_dft` command is not supported between the `read_scan_def` and `write_scan_def` commands and errors out. The netlist SCANDEF flow does not support a hierarchical approach; Design Compiler does not read multiple SCANDEF files.

This section contains the following subsections:

- [Identifying the Scan Chain Structure](#)
- [Performing Scan Chain Reordering in Design Compiler Graphical](#)
- [Writing Out the New Scan Chain Structure](#)

Identifying the Scan Chain Structure

Test structure includes scan chains as well as controlling logic. Controlling logic includes the scan enable, scan-in, scan-out, and test mode signals as well as other logic, such as compressors, decompressors, on-chip clocking (OCC), and memory BIST. The netlist SCANDEF flow identifies scan chains only.

Scan chains are identified by SCANDEF files. SCANDEF files can be generated by most DFT tools. [Example 12-4](#) shows an example of a SCANDEF file.

Example 12-4 SCANDEF File Example

```

VERSION 5.5 ;
NAMECASESENSITIVE ON ;
DIVIDERCHAR "/" ;
BUSBITCHARS "[]" ;
DESIGN TOP ;
SCANCHAINS 1 ;
- 1
+ START U20 o
+ FLOATING regA ( IN ti ) ( OUT q )
          regB ( IN ti ) ( OUT q )
          regF ( IN ti ) ( OUT q )
          regE ( IN ti ) ( OUT q )
+ ORDERED regC ( IN ti ) ( OUT q )
          U10 ( IN i_1 ) ( OUT o )
          regD ( IN d ) ( OUT q )
+ PARTITION clk_i_45_45
+ STOP regG ti ;

```

To identify all cells that are part of a scan chain, read the SCANDEF file using the `read_scan_def` command after you read the top-level design. The `read_scan_def` command performs the following tasks automatically:

- Reads the SCANDEF file
- Checks the SCANDEF file for consistency with the netlist

[Example 12-5](#) shows a `read_scan_def` consistency report. If the scan chains pass all structural checks, the Status column reports a V for VALIDATED. If a scan chain fails the structural checks, the Status column reports an F for FAILED. This example shows that the `top.scandef` SCANDEF file is consistent with the netlist.

Example 12-5 read_scan_def Consistency Report

```

dc_shell> read_scan_def top.scandef
Reading DEF file top.scandef
Checking SCANDEF...
Checking for duplication in SCANDEF...
Checking scan cell correspondence between SCANDEF and netlist...

```



```

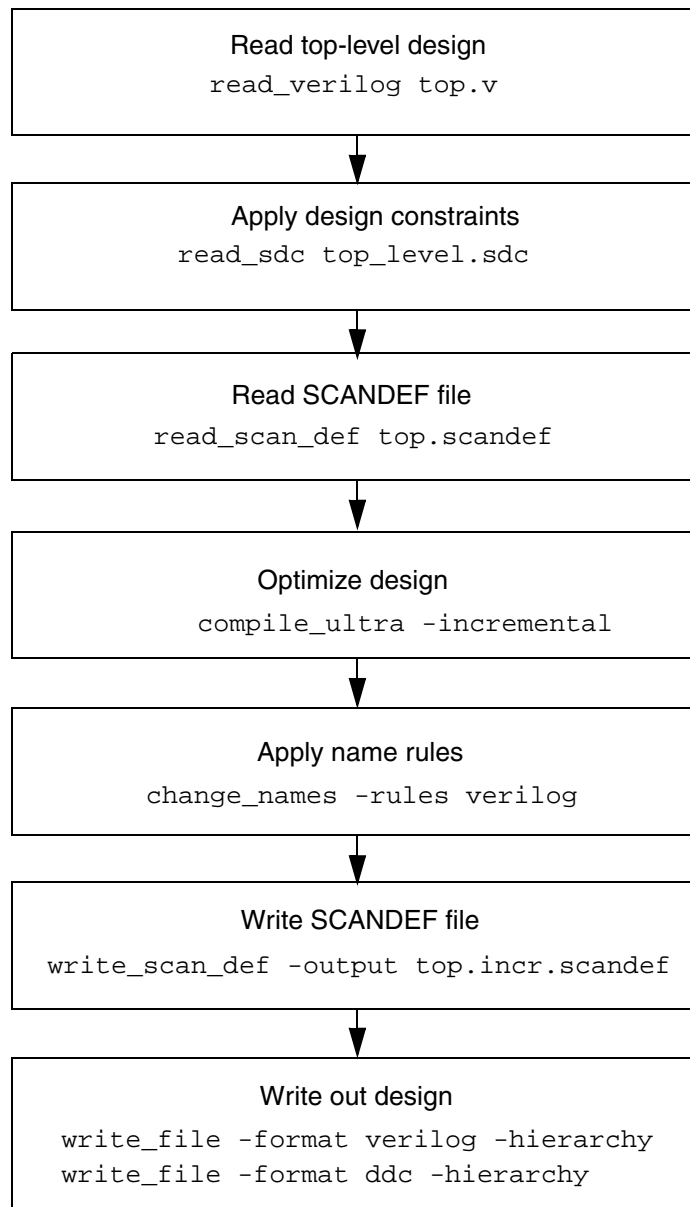
      Start scan chain checking...
      All checks completed.
*****
Report : Scan DEF check
Design : top
Version: E-2010.12
Date   : Fri Jul 23 04:54:38 2010
*****
Information from SCANDEF file:
  Number of SCANCHAINS: 3
Checking between SCANDEF file and design:
  Total SCANCHAINS checked: 3
  VALIDATED   :    3
  FAILED      :    0

Chain name Status  #cells PARTITION          Scan IN          Scan OUT
-----
1          V      2751  clk_45_45  scanin[1]  scanout[1]
2          V      2751  clk_45_45  scanin[2]  scanout[2]
3          V      2751  clk_45_45  scanin[3]  scanout[3]

```

Make sure that all scan chains are validated before you optimize the design. Any scan chains that fail are ignored and are not read in by the `read_scan_def` command.

Using the netlist SCANDEF flow, you can improve the timing of the functional logic while preserving the scan chain structure. [Figure 12-2](#) shows an example of a netlist SCANDEF flow. It is recommended that you use the `-incremental` option with the `compile_ultra` command. Other options, such as `-timing_high_effort_script` and `-spg`, are also supported. Be sure to use the appropriate design constraint commands for flows, such as UPF or multicorner-multimode.

Figure 12-2 Netlist SCANDEF Flow Example

For more information about the `read_scan_def` command, see the man page.

Performing Scan Chain Reordering in Design Compiler Graphical

In a Design Compiler Graphical flow, Synopsys physical guidance technology enables enhanced placement that is consistent with IC Compiler. The `-spg` option of the `compile_ultra` command enables this physical guidance technology.

In the netlist SCANDEF flow, you can perform an incremental compile using this physical guidance technology by using the following command:

```
dc_shell-topo> compile_ultra -spg -incremental
```

In this case, the tool further reduces scan chain wire length by performing scan chain reordering during the incremental compile. The reordering process honors any scan order constraints imposed by the input SCANDEF file.

Before running the incremental compile, you can use the `set_optimize_dft_options` command to configure scan reordering, just as you would in IC Compiler. For more information, see the man page.

Scan reordering during incremental compile is enabled by default when you use the `-spg` option. To disable it, set the `test_enable_scan_reordering_in_compile_incremental` variable to `false`.

Note the following requirements and limitations:

- Hierarchical SCANDEF is not supported; only a flat DFT flow is supported.
- Post-DFT DRC is not supported after reordering. Perform DRC checking in the ATPG tool to verify scan chain integrity.

Writing Out the New Scan Chain Structure

When you run the `write_scan_def` command, Design Compiler generates a SCANDEF file that contains the same scan chain structure as the input SCANDEF file. It traces the scan chains based on the SCANDEF file that was read by the `read_scan_def` command. If the `write_scan_def` is unable to trace a scan chain because the scan chain element cannot be found in the netlist, Design Compiler issues an error.

In general, you do not need to use the `-output` option with the `write_scan_def` command. If you omit the option, the SCANDEF file is included in the saved `.ddc` file for handoff to IC Compiler. However, the `-output` option is required if the handoff to IC Compiler is in ASCII format.

The output SCANDEF file contains all changes performed during the incremental optimization and `change_names` command specification steps.

The `write_scan_def` command does not support the `-expand_elements` option in the netlist SCANDEF flow because the `read_scan_def` command does not support hierarchical flows.

For more information about the `write_scan_def` command, see the man page.

13

Verifying Functional Equivalence

After optimization, you can use an equivalence checking tool to verify that your gate-level netlist is functionally equivalent to your RTL. This verification step ensures that the synthesis process or manual design changes did not introduce functional errors. You can use the Synopsys Formality tool or a third-party formal verification tool to perform functional equivalence checking.

This chapter contains the following sections:

- [Using Formality](#)
- [Adjusting Optimization for Successful Verification](#)
- [Using Third-Party Formal Verification Tools](#)

Using Formality

The Formality tool uses formal techniques to prove or disprove the functional equivalence of two designs. Formality performs RTL-to-RTL, RTL-to-gate, and gate-to-gate verifications. Functional equivalence checking does not take into account timing; it is a static verification process.

By default, Design Compiler automatically creates a Formality automated setup file in your working directory. This file has the extension .svf (setup verification file) and is named default.svf. The automated setup file provides a method for automatically conveying setup information to Formality. It alleviates the need to enter setup information manually, a task that can be time-consuming and error-prone.

The automated setup file is a binary file. When Formality reads this binary data, it will automatically convert it to ASCII and write it to a text file. This gives you the opportunity to review the setup information before you use it in the verification process.

The automated setup file can contain the following information:

- The settings of certain Tcl variables that affect how the names of design objects in the netlist are created, such as the `bus_naming_style` and `template_*` variables
- Optimizations that occur during RTL elaboration
- Name changes resulting from operations, such as `ungroup`, `group`, `uniquify`, `rename_design`, or the automatic `uniquify` process in `compile` with combinations of `group` and `ungroup` (these operations can change the names of design objects such as registers, black boxes, and top-level ports)
- Phase inversion and constant propagation optimizations performed during sequential mapping
- Datapath transformations, including information about multiplier architectures chosen during synthesis
- Retiming optimizations
- Information about FSM extraction

Additionally, the automated setup file records implicit `ungroup` operations. Implicit `ungroup` operations can occur in the following situations:

- During automatic ungrouping with the `compile_ultra` command, the `compile -ungroup -all` command, or the `compile -auto_ungroup` command
- If a design has an `ungroup` attribute set on it

- When DesignWare auto-ungroups DesignWare parts
- When certain user hierarchies are auto-ungrouped for datapath optimization

You can use the Design Compiler `set_svf` command to control the name of the automated setup file or to save it to a location other than your current working directory. To disable the generation of the .svf file, use the `-off` option. If you use the `set_svf` command, you must do so at the beginning of the synthesis process, before you read your design files.

The syntax of the `set_svf` command is

```
set_svf filename [-append] [-off]
```

For more information, see the *Formality User Guide*.

Adjusting Optimization for Successful Verification

Hard verifications occur when Formality cannot complete verification due to issues, such as design complexity. Datapath intensive designs or designs containing complex design components, such as parity generators, XOR trees, or very large cones of logic can cause Formality to issue inconclusive or hard verifications.

Design Compiler adjusts some of the optimizations to help reduce hard verifications, as discussed in the following sections. The first method, using the `set_verification_priority` command, is the recommended method for resolving inconclusive verification because it targets specific blocks or operators while minimizing QoR impact.

Using the `set_verification_priority` Command

The `set_verification_priority` command sets the `verification_priority` attribute on the specified objects. During the `compile_ultra` command run, the optimizations of the design with the `verification_priority` attribute are adjusted depending on the verification priority level so that the potential for hard verification is reduced. The `set_verification_priority` command is available only in DC Ultra.

When you have inconclusive verifications in Formality, you can use the `analyze_points` command to examine the hard points and determine the possible cause of the hard verifications. The `analyze_points` command runs analysis on the most recent failed or hard verification. The `analyze_points` command generates a recommendation as to which blocks or operators the `set_verification_priority` command can be used with in Design Compiler to reduce hard verifications.

You should apply the `set_verification_priority` commands after the design is read and elaborated.

To adjust optimization to reduce hard verifications,

1. Run Design Compiler.
2. Run Formality.
3. Analyze the Formality verification reports.

The `analyze_points` command runs analysis on the most recent failed or hard verification. The command generates a recommendation regarding which blocks the `set_verification_priority` command can be used with in Design Compiler to reduce hard verifications.

The following example shows the output of the `analyze_points` command in Formality:

```
Try adding the following command(s) to your Design Compiler script right
before the first compile_ultra command:
current_design test
set_verification_priority [ get_cells { add_21 mult_21 mult_30 sub_30 } ]
```

4. Rerun Design Compiler, as follows:
 - a. Specify the `set_verification_priority` command with the problem blocks identified by the `analyze_points` command in Formality. Add the commands that were suggested by Formality to your Design Compiler scripts.
 - b. Run the `compile_ultra` command.

For more information about the `analyze_points` command, see the man page in Formality and the *Formality User Guide*. For more information about the `set_verification_priority` command, see the man page in Design Compiler.

Using Single-Pass Verification

Design Compiler single-pass verification functionality adjusts optimization so that formal verification compatibility is prioritized over QoR. This is helpful if you do not have tight QoR constraints in your design, you do not have register retiming, and you want to pass formal verification with as little effort as possible. You can enable the single-pass verification flow by setting the `simplified_verification_mode` variable to `true`. The variable is set to `false` by default.

By default, designs with the `optimize_registers` attribute and `DW_div_pipe` DesignWare components are not retimed because they can adversely affect verification success. However, optimizing designs with the `optimize_registers` attribute and `DW_div_pipe` DesignWare components without retiming them can have a large impact on QoR and runtime. To enable the retiming of these designs, set the `simplified_verification_mode_allow_retiming` variable to `true` when you enable the `simplified_verification_mode` variable.

For more information about these variables, see the man pages in Design Compiler.

Using Third-Party Formal Verification Tools

To record setup information for formal verification tools other than Formality, use the `set_vsd` command.

The syntax is

```
set_vsd filename [-append] [-off]
```

The command records setup information in V-SDC format for efficient compare point matching in third-party formal verification tools. The V-SDC format is a subset of the automated setup file used by Formality. The V-SDC file is written in plain text, whereas the automated setup file is encrypted and compressed. The `set_vsd` command records the following operations:

- Name changes resulting from operations such as `ungroup`, `group`, or `uniquify`, or the automatic `uniquify` process in `compile` with combinations of `group` and `ungroup`. These operation can change the names of design objects such as registers, black boxes, and top-level ports
- Operations performed by the `compile` command that result in register optimizations.

14

Latch-Based Design Code Examples

This chapter provides code examples of designs that use various types of latches. It contains the following sections:

- [SR Latch](#)
- [D Latch](#)
- [D Latch With Asynchronous Reset](#)
- [D Latch With Asynchronous Set and Reset](#)
- [D Latch With Enable \(avoiding clock gating\)](#)
- [D Latch With Enable and Asynchronous Reset](#)
- [D Latch With Enable and Asynchronous Set](#)
- [D Latch With Enable and Asynchronous Set and Reset](#)

SR Latch

This section shows the VHDL and Verilog code that implements a design that uses an SR latch. It includes these subsections:

- [VHDL and Verilog Code Examples for SR Latch](#)
- [Inference Report for an SR Latch](#)
- [Synthesized Design for an SR Latch](#)

VHDL and Verilog Code Examples for SR Latch

To implement an SR latch in either VHDL or Verilog, you must set the following variable to true:

```
hdlin_report_inferred_modules
```

You must also set the following attribute:

```
attribute async_set_reset of RESET, SET : signal is "true";
```

[Example 14-1](#) shows the VHDL code that infers an SR latch.

Example 14-1 VHDL Code for an SR Latch

```
library IEEE, SYNOPSYS;
use IEEE.std_logic_1164.all;
use SYNOPSYS.attributes.all;

entity SR_LATCH is
  port ( RESET, SET : in std_logic;
        Y : out std_logic );
end SR_LATCH;

architecture BEHAVIORAL of SR_LATCH is
  attribute async_set_reset of RESET, SET : signal is "true";
begin
  infer : process ( RESET, SET )
  begin
    if ( RESET = '0' ) then
      y <= '0';
    elsif ( SET = '0' ) then
      y <= '1';
    end if;
  end process infer;
end BEHAVIORAL;
```

[Example 14-2](#) shows Verilog code that infers an SR latch.

Example 14-2 Verilog Code Example for an SR Latch

```
module SR_LATCH( reset,set, y);
input reset,set ;
output y ;
// synopsys async_set_reset "reset,set"
reg y ;

always @(set or reset)
begin : infer
if (reset == 0)
y = 1'b0 ;
else if (set == 0)
y = 1'b1 ;
end
endmodule
```

Inference Report for an SR Latch

[Example 14-3](#) shows the inference report generated for an SR latch from the VHDL code shown in [Example 14-1](#) or the Verilog code in [Example 14-2](#).

Example 14-3 Inference Report for an SR Latch

```
Inferred memory devices in process 'infer'
in routine SR_LATCH line 13 in
file '/home/sudipto/work/latch_appl/rtl/vhdl/sr_latch.vhdl'.
```

```
=====
```

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Y_reg	Latch	1	-	-	Y	Y	-	-	-

```
=====
```

```
Y_reg
-----
Async-reset: RESET'
Async-set: SET'
Async-set and Async-reset ==> Q: 0
```

Synthesized Design for an SR Latch

Figure 14-1 shows the synthesized design for the SR-latch-based design resulting from compilation of the code shown in [Example 14-1](#) or [Example 14-2](#). In this synthesis, LSR0 is an SR latch and both S and R are active-low Inputs. Here is the target library description of the latch for the cell LSR0:

```
latch ("IQ","IQN") {  
  clear    : "R'";  
  preset   : "S'";  
  clear_preset_var1 : L;  
  clear_preset_var2 : L;  
}
```

Figure 14-1 Synthesized Design for an SR Latch



D Latch

This section shows the VHDL code that implements a design that uses a simple D latch. It includes these subsections:

- [VHDL Code for a D Latch](#)
- [Inference Report for a D Latch](#)
- [Synthesized Design for a D Latch](#)

VHDL Code for a D Latch

To implement a D latch in VHDL, you must set the following variable to true:

```
hdlin_report_inferred_modules
```

[Example 14-4](#) shows the VHDL code that implements a design using a simple D latch.

Example 14-4 VHDL Code for a D Latch

```
library IEEE, SYNOPSYS;
use IEEE.std_logic_1164.all;
use SYNOPSYS.attributes.all;

entity d_latch is
  port ( enable, data : in std_logic;
        y : out std_logic );
end d_latch;

architecture behavioral of d_latch is
begin
  infer : process ( enable, data )
  begin
    if ( enable = '1' )
    then
      y <= data;
    end if;
  end process infer;
end behavioral;
```

Inference Report for a D Latch

[Example 14-5](#) shows the inference report for a D latch resulting from compilation of the code in [Example 14-4](#).

Example 14-5 Inference Report for a D Latch

```
Inferred memory devices in process 'infer'
  in routine d_latch line 13 in file
    '/home/sudipto/work/latch_appl/rtl/vhdl
    /d_latch.vhdl'.
```

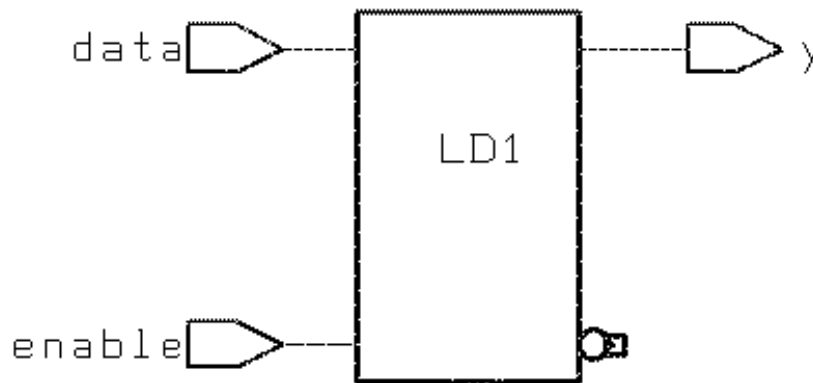
```
=====
| Register Name | Type | Width | Bus | MB | AR | AS | SR | SS | ST |
=====
| y_reg         | Latch | 1 | - | - | N | N | - | - | - |
=====
y_reg
-----
reset/set: none
```

Synthesized Design for a D Latch

Figure 14-2 shows the synthesized design for the D latch resulting from compilation of the code in Example 14-4. In this design, LD1 is the simple D latch. Here is the target library description of the latch for the cell LD1:

```
latch ("IQ","IQN") {  
  enable : "G";  
  data_in : "D";  
}
```

Figure 14-2 Synthesized Design for a D Latch



D Latch With Asynchronous Reset

This section shows the VHDL and Verilog code that implements a design that uses a D latch with asynchronous reset. It includes these subsections:

- [VHDL and Verilog Code for a D Latch With Asynchronous Reset](#)
- [Inference Report for a D Latch With Asynchronous Reset](#)
- [Synthesized Design for a D Latch With Asynchronous Reset](#)

VHDL and Verilog Code for a D Latch With Asynchronous Reset

To implement a D latch with asynchronous reset in either VHDL or Verilog, you must set the following variable to true:

```
hdlin_report_inferred_modules
```


You must also set the following attribute, as illustrated by the code examples:

```
attribute async_set_reset of RESET, SET : signal is "true";
```

[Example 14-6](#) shows VHDL code for a D latch with asynchronous reset.

Example 14-6 *VHDL Code for a D Latch With Asynchronous Reset*

```
library IEEE, SYNOPSYS;
use IEEE.std_logic_1164.all;
use SYNOPSYS.attributes.all;

entity d_latch_async_reset is
  port ( enable, reset, data : in std_logic;
        q : out std_logic );
end d_latch_async_reset;

architecture behavioral of d_latch_async_reset is
  attribute async_set_reset of reset : signal is "true";
begin
  infer : process ( enable, reset, data )
  begin
    if ( reset = '1' )
    then
      q <= '0';
    elsif ( enable = '1' )
    then
      q <= data;
    end if;
  end process infer;
end behavioral;
```

[Example 14-7](#) shows Verilog code for the D latch with asynchronous reset.

Example 14-7 *Verilog Code for a D Latch With Asynchronous Reset*

```
module d_latch_async_reset (enable, reset, data, q) ;
input enable, data, reset ;
output q ;
// synopsys async_set_reset "reset"
reg q ;

always @(reset or enable or data)
begin : infer
  if (reset == 1)
    q = 1'b0 ;
  else if (enable == 1)
    q = data ;
  end
endmodule
```

Inference Report for a D Latch With Asynchronous Reset

[Example 14-8](#) shows the inference report for a D latch with asynchronous reset resulting from compilation of the code in [Example 14-6](#) or [Example 14-7](#).

Example 14-8 Inference Report for a D Latch With Asynchronous Reset

```
Inferred memory devices in process 'infer'
  in routine d_latch_async_reset
    line 13 in file
      '/home/sudipto/work/latch_appl/rtl/vhdl/d_latch_async_reset.vhdl'.
=====
| Register Name | Type | Width | Bus | MB | AR | AS | SR | SS | ST |
=====
|      q_reg    | Latch |    1  | -   | -  | Y  | N  | -  | -  | -  |
=====

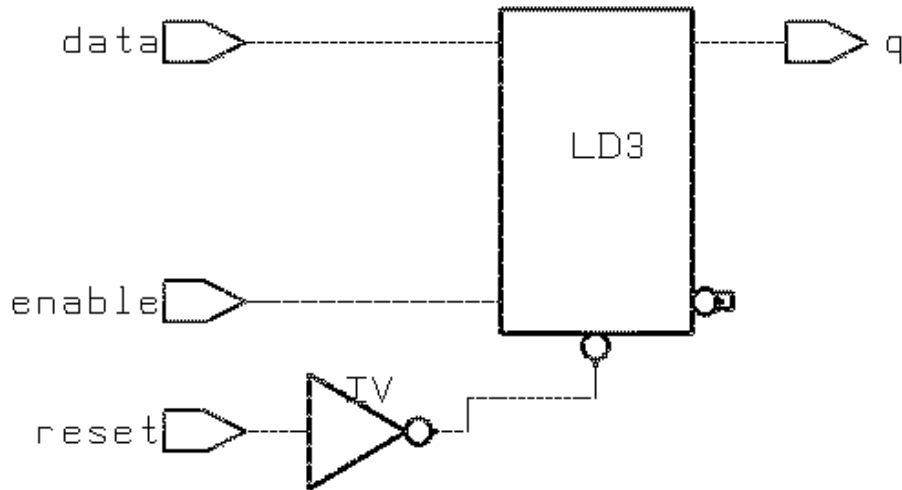
q_reg
-----
  Async-reset: reset
```

Synthesized Design for a D Latch With Asynchronous Reset

[Figure 14-3](#) shows the synthesized design for the D latch resulting from compilation of the code shown in [Example 14-6](#) and [Example 14-7](#). In this design, LD3 is the simple D latch. Here is the target library description of the latch for the cell LD3:

LD3 is a D latch with asynchronous active-low reset (CD). The description of the latch for the cell LD3 in the target library is as follows:

```
latch ("IQ", "IQN") {
  enable   : "G";
  data_in  : "D";
  clear    : "CD' ";
}
```

Figure 14-3 Synthesized Design for a D Latch With Asynchronous Reset

D Latch With Asynchronous Set and Reset

This section shows the VHDL and Verilog code that implements a design that uses a D latch with asynchronous set and reset. It includes these subsections:

- [VHDL and Verilog Code for a D Latch With Asynchronous Set and Reset](#)
- [Inference Report for a D Latch With Asynchronous Set and Reset](#)
- [Synthesized Design for a D Latch With Asynchronous Set and Reset](#)

VHDL and Verilog Code for a D Latch With Asynchronous Set and Reset

To implement a D latch with asynchronous set and reset in either VHDL or Verilog, you must set the following variable to true:

```
hdlin_report_inferred_modules
```

You must also set the following attributes, as illustrated by the code examples:

```
attribute async_set_reset of set, reset : signal is "true";
attribute one_hot of set, reset : signal is "true";
```

[Example 14-9](#) shows the VHDL code for a D latch with asynchronous set and reset attributes.

Example 14-9 VHDL Code for a D Latch With Asynchronous Set and Reset

```
library IEEE, SYNOPSYS;
use IEEE.std_logic_1164.all;
use SYNOPSYS.attributes.all;

entity d_latch_async_set_reset is
  port ( enable, set, reset, data : in std_logic;
        q : out std_logic );
end d_latch_async_set_reset;

architecture behavioral of d_latch_async_set_reset is
  attribute async_set_reset of set, reset : signal is "true";
  attribute one_hot of set, reset : signal is "true";
begin
  infer : process ( enable, set, reset, data )
  begin
    if ( reset = '1' )
    then
      q <= '0';
    elsif ( set = '1' )
    then
      q <= '1';
    elsif ( enable = '1' )
    then
      q <= data;
    end if;
  end process infer;
end behavioral;
```

[Example 14-10](#) shows the Verilog code for a D latch with asynchronous set and reset attributes.

Example 14-10 Verilog Code for a D Latch With Asynchronous Set and Reset

```
module d_latch_async_set_reset
  (enable, set, reset, q, data) ;

  input enable, set, reset, data ;
  output q ;

  // synopsys async_set_reset "set, reset"
  // synopsys one_hot "set, reset"

  reg q ;

  always @(enable or set or reset or data)
    begin : infer
      if (reset == 1)
        q = 1'b0 ;
      else if (set == 1)
        q = 1'b1 ;
      else if (enable == 1)
        q = data ;
    end
endmodule
```

Inference Report for a D Latch With Asynchronous Set and Reset

[Example 14-11](#) shows the inference report for a D latch with asynchronous set and reset resulting from compilation of the code shown in [Example 14-9 on page 14-10](#) or [Example 14-10](#).

Example 14-11 Inference Report for a D Latch With Asynchronous Set and Reset

```
Inferred memory devices in process 'infer'
  in routine d_latch_async_set_reset
    line 14 in file
      '/home/sudipto/work/latch_appl/rtl/vhdl/d_latch_async_set_reset.vhdl'.
=====
|   Register Name   |   Type | Width | Bus | MB | AR | AS | SR | SS | ST |
=====
|       q_reg       |  Latch |    1  |  -  | -  | Y  | Y  | -  | -  | -  |
=====

q_reg
----
  Async-reset: reset
  Async-set: set
  Async-set and Async-reset ==> Q: X
```

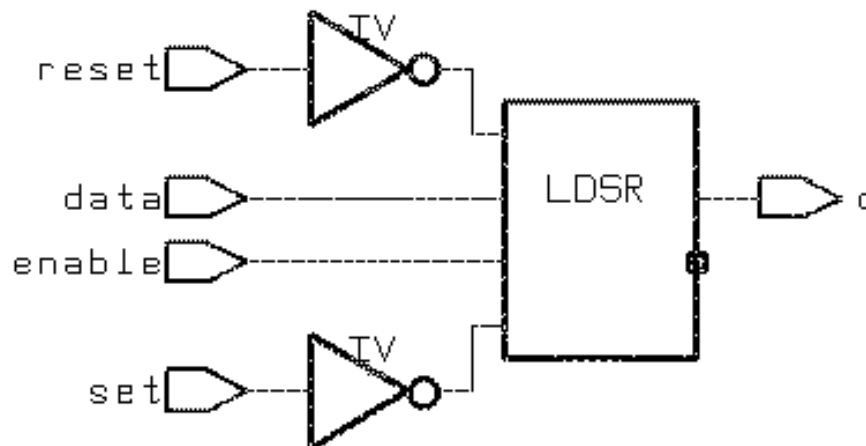
Synthesized Design for a D Latch With Asynchronous Set and Reset

Figure 14-4 shows the synthesized design for the D latch resulting from compilation of the code in Example 14-9 and Example 14-10. In this design, LDSR is a D latch with active-low asynchronous set (SET) and reset (CLR).

The description of the latch for the cell LDSR in the target library is as follows:

```
latch ("IQ","IQN") {
  enable   : "G";
  data_in  : "D";
  clear    : "CLR'";
  preset   : "SET'";
  clear_preset_var1 : L;
  clear_preset_var2 : L;
}
```

Figure 14-4 Synthesized Design for a D Latch With Asynchronous Set and Reset



D Latch With Enable (avoiding clock gating)

This section shows the VHDL and Verilog code that implements a design that uses a D latch with the enable attribute to avoid clock gating. It includes these subsections:

- [VHDL and Verilog Code for a D Latch With Enable](#)
- [Inference Report for a D Latch With Enable](#)
- [Synthesized Design for a D Latch With Enable](#)
- [Inferring Gated Clocks](#)

VHDL and Verilog Code for a D Latch With Enable

To implement a D latch with enable in either VHDL or Verilog, you must set the following variable to true:

```
hdlin_report_inferred_modules
```

You must also set the following attribute to true:

```
hdlin_keep_feedback
```

[Example 14-12](#) shows the VHDL code for a D latch with enable.

Example 14-12 VHDL Code for a D Latch With Enable

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity d_latch_enab is
  port ( enable, clock, data : in std_logic;
        q : buffer std_logic );
end d_latch_enab;

architecture behavioral of d_latch_enab is
begin
  infer : process ( enable, clock, data )
  begin
    if ( clock = '1' )
    then
      if ( enable = '1' )
      then
        q <= data;
      else
        q <= q;
      end if;
    end if;
  end process infer;
end behavioral;
```

[Example 14-13](#) shows the Verilog code for a D latch with enable.

Example 14-13 Verilog Code for a D Latch With Enable

```
module d_latch_enab ( enable, clock, data, q) ;

input enable, clock, data ;
output q ;

reg q ;

always @(enable or clock or data)
begin :infer
    if (clock == 1)
    begin
        if (enable == 1)
            q = data ;
        else
            q = q ;
        end
    end
end
endmodule
```

Inference Report for a D Latch With Enable

[Example 14-14](#) shows the inference report for a D latch with enable resulting from compilation of the code in [Example 14-12 on page 14-13](#) or [Example 14-13](#).

Example 14-14 Inference Report for a D Latch With Enable

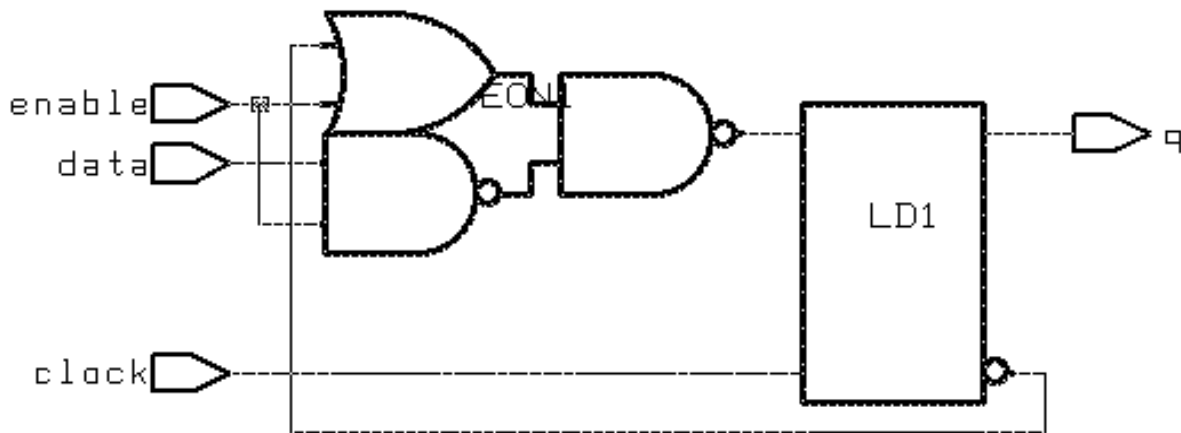
```
Inferred memory devices in process 'infer'
in routine d_latch_enab line 13 in
file '/home/sudipto/work/latch_appl/rtl/vhdl/d_latch_enab.vhdl'.
=====
| Register Name | Type | Width | Bus | MB | AR | AS | SR | SS | ST |
|=====|
| q_reg        | Latch | 1 | - | - | N | N | - | - | - |
|=====|

q_reg
-----
reset/set: none
```


Synthesized Design for a D Latch With Enable

Figure 14-5 shows the synthesized design for the D latch with enable resulting from compilation of the code in [Example 14-12](#) or [Example 14-13](#).

Figure 14-5 Synthesized Design for a D Latch With Enable



Inferring Gated Clocks

This section describes two cases—Case 1 and Case 2—in which HDL Compiler infers gated clocks.

Case 1

If the variable `hdlin_keep_feedback` is not set to true, then HDL Compiler assumes the default value of false and removes all feedback loops. For example, feedback loops inferred from a statement such as the following

```
Q = Q
```

are removed.

The loop that is inferred from the following statement, shown in VHDL code, is removed.

```

if ( enable = '1' )
then
  q<= data;
else
  q <= q;
end if;

```

The code indicates that the gated clock in the synthesized design in [Figure 14-6](#), which does not have a feedback loop, is removed.

Case 2

Gated clocks can also be inferred from the coding style used to implement a design. For example, if the VHDL code is written in either of the coding styles in [Example 14-15](#) or [Example 14-16](#), regardless of whether `hdlin_keep_feedback` is set to true, Design Compiler will create a gated clock for the design.

[Example 14-15](#) implies a priority coding style— that is, the clock value is assessed first and enable is considered only if the clock is a certain value.

Example 14-15 Coding Style A

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity d_latch_enab is
  port ( enable, clock, data : in std_logic;
        q : out std_logic );
end d_latch_enab;

architecture behavioral of d_latch_enab is
begin
  infer : process ( enable, clock, data )
  begin
    if ( clock = '1' )
    then
      if ( enable = '1' )
      then
        q <= data;
      end if;
    end if;
  end process infer;
end behavioral;

```

For [Example 14-16](#), no priority is implied.

Example 14-16 Coding Style B

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

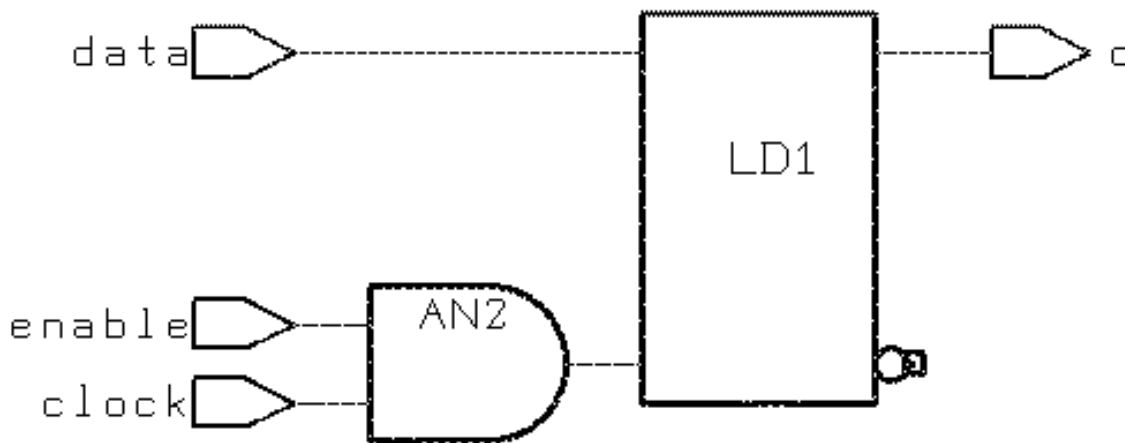
entity d_latch_enab is
  port ( enable, clock, data : in std_logic;
        q : out std_logic );
end d_latch_enab;

architecture behavioral of d_latch_enab is
begin
  infer : process ( enable, clock, data )
  begin
    if ( clock = '1' and enable = '1')
      q <= data;
    end if;
  end process infer;
end behavioral;
```

Synthesized Design With Enable and Gated Clock

[Figure 14-6](#) shows the synthesized design for the D latch with enable and clock gating resulting from compilation of the code in [Example 14-15 on page 14-16](#) and [Example 14-16](#).

Figure 14-6 D Latch With Enable and Gated Clock



D Latch With Enable and Asynchronous Reset

This section shows the VHDL and Verilog code that implements a design that uses a D latch with the enable and asynchronous reset attributes. It includes these subsections:

- [VHDL and Verilog Code for a D Latch With Enable and Asynchronous Reset](#)
 - [Synthesized Design for a D Latch With Enable and Asynchronous Reset](#)
-

VHDL and Verilog Code for a D Latch With Enable and Asynchronous Reset

To implement a D latch with enable and asynchronous reset in either VHDL or Verilog, you must set the following variables to true:

```
hdlin_report_inferred_modules
hdlin_keep_feedback
```

You must also set the following attribute:

```
attribute async_set_reset of reset : signal is "true";
```

[Example 14-17](#) shows the VHDL code for a D latch with enable and asynchronous reset.

Example 14-17 VHDL Code for a D Latch With Enable and Asynchronous Reset

```
library IEEE, SYNOPSYS;
use IEEE.STD_LOGIC_1164.all;
use SYNOPSYS.attributes.all;

entity d_latch_enab_async_reset is
  port ( enable, clock, reset, data : in std_logic;
        q : buffer std_logic );
end d_latch_enab_async_reset;

architecture behavioral of d_latch_enab_async_reset is
  attribute async_set_reset of reset : signal is "true";
begin
  infer : process ( enable, clock, reset, data )
    variable temp : std_logic;
  begin
    temp := q;
    if ( reset = '1' ) then
      q <= '0';
    elsif ( clock = '1' ) then
      case enable is
        when '1' => q <= data;
        when others => q <= temp;
      end case;
    end if;
  end infer;
end behavioral;
```

```
    end process infer;
end behavioral;
```

[Example 14-18](#) shows the Verilog code for a D latch with enable and asynchronous reset.

Example 14-18 *Verilog Code for a D Latch With Enable and Asynchronous Reset*

```
module d_latch_enab_async_reset
    (enable, clock, reset, q, data) ;

    input enable, clock, reset, data ;
    output q ;

    // synopsys async_set_reset "reset"

    reg q ;

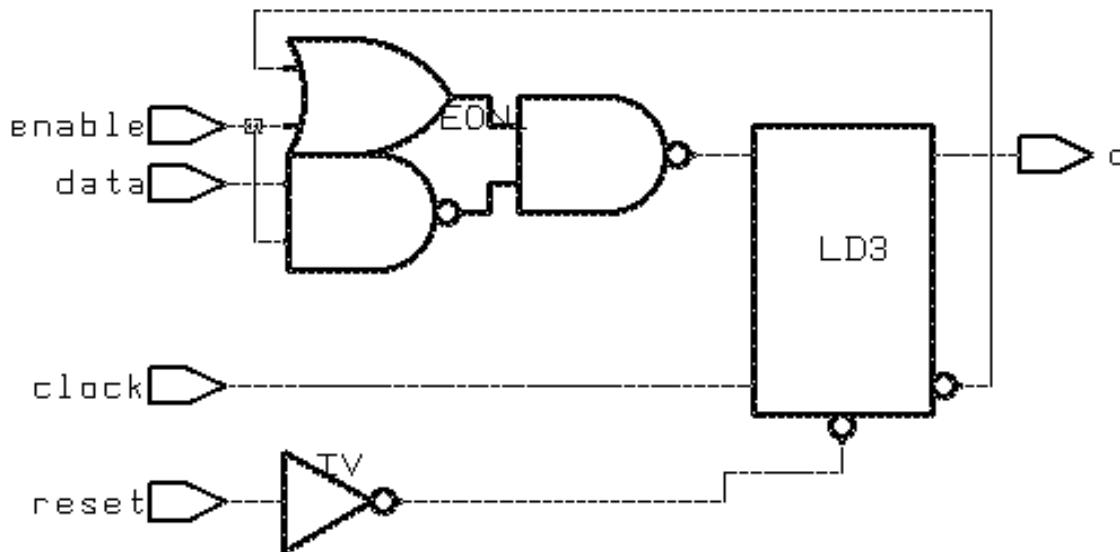
    always @(enable or clock or reset or data)
        begin : infer
            if (reset == 1)
                q = 1'b0 ;

            else if (clock == 1)
                begin
                    if (enable == 1)
                        q = data ;
                    else
                        q = q ;
                end
            end
        end
    endmodule
```

Synthesized Design for a D Latch With Enable and Asynchronous Reset

Figure 14-7 shows the synthesized design for the D latch with enable and asynchronous reset resulting from compilation of the code in [Example 14-17](#) or [Example 14-18](#).

Figure 14-7 Synthesized Design for a D Latch With Enable and Asynchronous Reset



D Latch With Enable and Asynchronous Set

This section shows the VHDL and Verilog code that implements a design that uses a D latch with the enable and asynchronous set attributes. It includes these subsections:

- [VHDL and Verilog Code for a D Latch With Enable and Asynchronous Set](#)
- [Synthesized Design for D Latch With Enable and Asynchronous Set](#)

VHDL and Verilog Code for a D Latch With Enable and Asynchronous Set

To implement a D latch with enable and asynchronous set in either VHDL or Verilog, you must set the following variables to true:

```
hdlin_report_inferred_modules
hdlin_keep_feedback
```

You must also set the following attribute:

```
attribute async_set_reset of set : signal is "true";
```

Example 14-19 shows the VHDL code for a D latch with enable and asynchronous reset.

Example 14-19 *VHDL Code for D Latch With Enable and Asynchronous Set*

```
library IEEE, SYNOPSYS;
use IEEE.STD_LOGIC_1164.all;
use SYNOPSYS.attributes.all;

entity d_latch_enab_async_set is
  port ( enable, clock, set, data : in std_logic;
        q : buffer std_logic );
end d_latch_enab_async_set;

architecture behavioral of d_latch_enab_async_set is
  attribute async_set_reset of set : signal is "true";
begin
  infer : process ( enable, clock, set, data )
  begin
    if ( set = '1' )
    then
      q <= '1';
    elsif ( clock = '1' )
    then
      if ( enable = '1' )
      then
        q<= data;
      else
        q <= q;
      end if;
    end if;
  end process infer;
end behavioral;
```

[Example 14-20](#) shows the Verilog code for a D latch with enable and asynchronous set.

Example 14-20 Verilog Code for D Latch With Enable and Asynchronous Set

```
module d_latch_enab_async_set (enable, clock, set, q, data) ;

input enable, clock, set, data ;
output q ;

// synopsys async_set_reset "set"

reg q ;

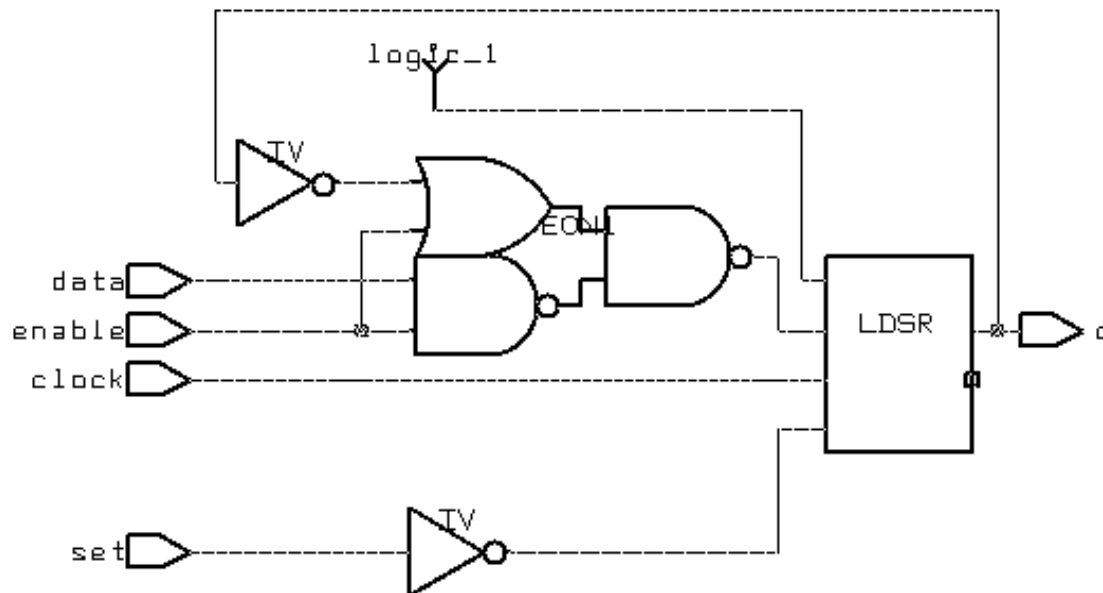
always @(enable or clock or set or data)
begin : infer
    if (set == 1)
        q = 1'b1 ;

        else if (clock == 1)
            begin
                if (enable == 1)
                    q = data ;
                else
                    q = q ;
            end
        end
end
endmodule
```


Synthesized Design for D Latch With Enable and Asynchronous Set

Figure 14-8 shows the synthesized design for the D latch with enable and asynchronous set resulting from compilation of the code shown in [Example 14-19](#) or [Example 14-20](#).

Figure 14-8 Synthesized Design for D Latch With Enable and Asynchronous Set



D Latch With Enable and Asynchronous Set and Reset

This section shows the VHDL and Verilog code that implements a design that uses a D latch with the enable and asynchronous set and reset attributes. It includes these subsections:

- [VHDL and Verilog Code for D Latch With Enable and Asynchronous Set and Reset](#)
- [Synthesized Design for D Latch With Enable and Asynchronous Set and Reset](#)

VHDL and Verilog Code for D Latch With Enable and Asynchronous Set and Reset

To implement a D latch with enable and asynchronous set and reset in either VHDL or Verilog, you must set the following variables to true:

```
hdlin_report_inferred_modules
hdlin_keep_feedback
```

You must also set the following attributes:

```
attribute async_set_reset of set : signal is "true";
attribute one_hot of set, reset: signal is "true";
```

Example 14-21 shows the VHDL code for a D latch with enable and asynchronous set and reset.

Example 14-21 *VHDL Code for D Latch With Enable and Asynchronous Set and Reset*

```
library IEEE, SYNOPSYS;
use IEEE.STD_LOGIC_1164.all;
use SYNOPSYS.attributes.all;

entity d_latch_enab_async_set_reset is
  port ( enable, clock, set, reset, data : in std_logic;
        q : buffer std_logic );
end d_latch_enab_async_set_reset;

architecture behavioral of d_latch_enab_async_set_reset is
  attribute async_set_reset of set, reset : signal is "true";
  attribute one_hot of set, reset : signal is "true";
begin
  infer : process (enable, clock, set, reset, data)
  begin
    if ( set = '1' ) then
      q <= '1';
    elsif ( reset = '1' ) then
      q <= '0';
    elsif ( clock = '1' ) then
      if ( enable = '1' ) then
        q <= data;
      else
        q <= q;
      end if;
    end if;
  end process infer;
end behavioral;
```

[Example 14-22](#) shows the Verilog code for a D latch with enable and asynchronous set and reset.

Example 14-22 Verilog Code for D Latch With Enable and Asynchronous Set and Reset

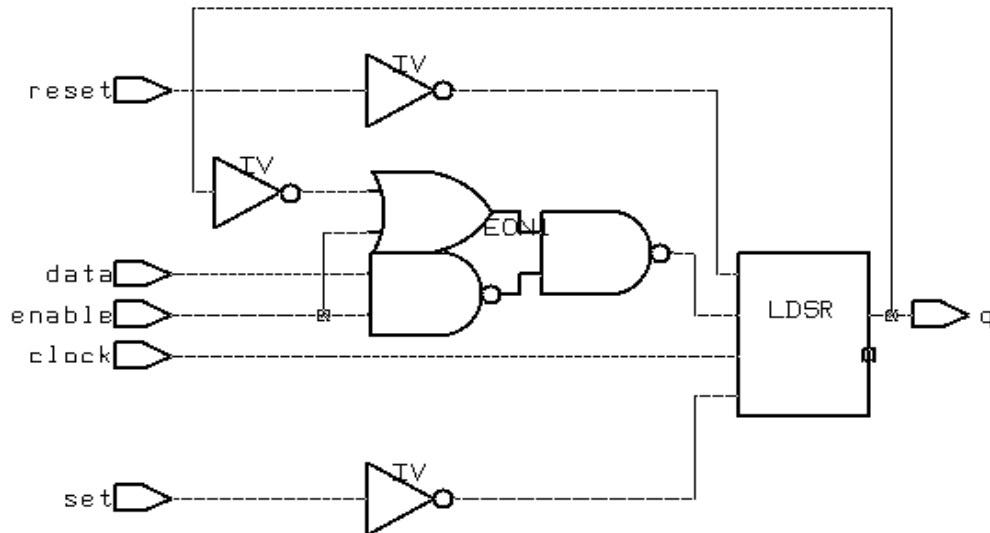
```
module d_latch_enab_async_set_reset (enable, clock, set,
reset, q, data);
input enable, clock, set, reset, data ;
output q ;
// synopsys async_set_reset "set, reset"
// synopsys one_hot "set, reset"
reg q ;

always @(enable or clock or set or reset or data)
begin : infer
    if (reset == 1)
        q = 1'b0 ;
    else if (set == 1)
        q = 1'b1 ;
    else if (clock == 1)
        begin
            if (enable == 1)
                q = data ;
            else
                q = q ;
        end
    end
end
endmodule
```

Synthesized Design for D Latch With Enable and Asynchronous Set and Reset

Figure 14-9 shows the synthesized design for the D latch with enable and asynchronous set and reset resulting from compilation of the code shown in [Example 14-21](#) or [Example 14-22](#).

Figure 14-9 Synthesized Design for D Latch With Enable and Asynchronous Set and Reset



Index

A

- analyzing datapath extraction 5-15
- architecture of an FSM 8-3
- area
 - exploring design space 1-3
- ASCII flow
 - incremental flow 12-1
 - incremental flow diagram 12-2
- assign statements in gate-level netlist,
preventing 3-21
- asynchronous reset example
 - D latch 14-6
 - D latch with enable and 14-18
- asynchronous set and reset example
 - D latch 14-9
 - D latch with enable and 14-23
- asynchronous set example
 - D latch with enable with 14-20
- attributes
 - clocked_on_also 3-24
 - dont_touch_network 3-16
 - fix_multiple_port_nets 3-21
 - multibit_width 7-7
- auto encoding style for FSMs 8-5
- auto_ungroup_preserve_constraints variable
4-9
- automatic compile flow for FSMs 8-7

- automatic methodology for FSMs 8-9

B

- balance_buffer command 3-6
- balanced buffering 3-22
- binary encoding style for FSMs 8-5
- bus_multiple_separator_style variable 7-5
- bus_range_separator_style variable 7-5

C

- capacitance
 - cost calculation 11-3
- case statements
 - used in multiplexing logic 6-1
- cells
 - ungroup 3-13
- clock network
 - preserving after clock tree synthesis 3-16
- clock network latency
 - multiple clocks per register 3-15
- clock pins, slave 3-24
- clock tree synthesis
 - preserving clock network 3-16
- clocked_on_also attribute 3-24

- clocks
 - multiple clocks per register
 - optimization 3-14
 - unattached slave 3-24
- commands 13-2
 - balance_buffer 3-6
 - compile
 - map_effort high option 3-12
 - compile_ultra 2-3
 - create_clock 3-14, 11-4
 - create_generated_clock 3-15
 - create_multibit 7-4
 - group 3-6
 - group_fsm 8-7
 - group_path 11-4
 - optimize_registers 1-13
 - read_scan_def 12-6
 - remove_isolate_ports 3-11
 - remove_multibit 7-4, 7-6
 - report_cell 9-7
 - report_design 3-19, 9-7
 - report_fsm 8-7, 8-13
 - report_isolate_ports 3-11
 - report_multibit 7-5
 - report_path_group 11-5
 - report_port 3-19
 - report_timing 3-12
 - report_transitive_fanout 3-16
 - report_wire_load 3-19
 - set_case_analysis 3-14
 - set_clock_latency 3-15
 - set_cost_priority 11-8
 - set_critical_range 3-5
 - set_disable_timing 3-14
 - set_dont_touch_network 3-16
 - set_dont_use 7-7
 - set_fix_multiple_port_nets 3-21
 - set_fsm_encoding 8-4, 8-7, 8-13
 - set_fsm_encoding_style 8-5, 8-7, 8-13
 - set_fsm_minimize 8-7
 - set_fsm_order 8-7
 - set_fsm_preserve_state 8-7
 - set_fsm_state_vector 8-4, 8-7, 8-13
 - set_isolate_ports 3-9
 - set_max_area
 - ignore_tns option 3-8
 - set_min_library 3-20
 - set_operating_conditions 3-20
 - set_register_type 9-6
 - set_scan_configuration 9-20, 9-21, 9-24
 - set_ungroup 3-13
 - set_vsdcc 13-5
 - ungroup 3-13
 - write_scan_def 12-9
- compilation
 - top level 2-8
- compilation strategy 2-11
 - compile-characterize-write script-recompile 2-13
 - how to determine 2-9
 - top-down hierarchical compile 2-10
- compile
 - full 1-9
 - high effort 3-7
 - incremental 3-8
 - log 2-17
 - test-ready 9-24
 - variables 2-5
- compile command
 - auto_ungroup area option 3-4
 - incompatible options of 2-8
 - incremental_mapping option 2-6
 - limitations 2-9
 - map_effort high option 3-12
- compile cost function 11-2
- compile flow chart 1-10
- compile log
 - customizing 11-10
- compile_auto_ungroup_override_wlm 4-10
- compile_delete_unloaded_sequential_cells variable 9-9
- compile_enable_register_merging variable 9-12

- compile_log_format variable 11-10
- compile_seqmap_propagate_constants variable 9-10
- compile_seqmap_propagate_high_effort variable 11-12
- compile_ultra command 2-3
- compile_variables command 2-5
- compiles
 - list of variables 2-5
- completely and incompletely specified FSMs 8-6
- components
 - multibit 7-1
 - removing 7-4, 7-6
- congestion
 - routing 7-7
- constraints
 - commands, list of 3-18
 - design rule 11-2
 - optimization 11-4
 - priorities
 - set_cost_priority command 11-8
 - violation fixing 2-8
- conventions for documentation xviii
- cost calculation
 - capacitance 11-3
 - fanout 11-3
 - maximum delay 11-4
 - minimum delay 11-6
 - transition time 11-3
- cost function 1-1, 2-1, 11-1, 11-2
 - design rule constraints 1-1, 2-1, 11-1, 11-2
 - optimization constraints 1-1, 2-1, 11-1, 11-2
- create_clock command
 - and path groups 11-4
 - multiple clocks per register 3-14
- create_generated_clock command
 - multiple clocks per register 3-15
- create_multibit command 7-4
- critical negative slack, defined 11-5

- critical path
 - defined 3-12
 - resynthesis 3-12
- critical range, defined 11-5
- critical-path resynthesis 3-7
- customer support xix

D

- D latch example 14-4
- D latch with asynchronous reset example 14-6
- D latch with asynchronous set and reset example 14-9
- D latch with enable and asynchronous reset example 14-18
- D latch with enable and asynchronous set and reset example 14-23
- D latch with enable and asynchronous set example 14-20
- D latch with enable example 14-13
- data path logic, synthesis of 7-1
- datapath extraction
 - DC Ultra 5-10
- datapath extraction, analyzing 5-15
- DC Ultra datapath optimization
 - datapath extraction 5-10
 - datapath report 5-13
 - licenses required 5-9
- default.svf file 13-2
- definitions
 - critical path 3-12
 - critical range 11-5
 - negative slack
 - critical 11-5
 - total 11-6
 - worst 11-4
- delay
 - optimize for 3-3
- delay analysis
 - control and report 3-20

- delay cost, calculating
 - maximum 11-4
 - minimum 11-6
- design rule
 - constraints 11-2
- design rule fixing
 - control 2-7
- design rule violations
 - fixing 2-9
- design space
 - exploring speed and area 1-3
- design space curve 1-4
- designs
 - exploring design space 1-3
 - space curve 1-4
 - ungroup 3-13
- DesignWare library
 - specifying 5-5
- don't care conditions for FSMs 8-2
- dont_touch_network attribute 3-16

E

- exploring design space 1-3

F

- fanin gates
 - mapping to wide- 3-13
- fanout load
 - cost calculation 11-3
- feedthrough nets, eliminating 3-21
- files
 - compile log 2-17
- finite state machine
 - architecture 8-3
 - auto encoding style 8-5
 - automatic flow compile process 8-9
 - basic description 8-2
 - binary encoding style 8-5

- completely and incompletely specified FSMs 8-6
- DC Ultra automatic compile flow 8-7, 8-11
 - supported commands 8-7
- DC Ultra automatic methodology 8-9
- design file requirements 8-8
- don't care condition 8-2
- FSM 8-1
 - gray encoding style 8-5
 - Mealy machine 8-3
 - Moore machine 8-3
 - one-hot encoding style 8-5
 - reports 8-13
 - state assignment 8-2
 - state encoding styles 8-3, 8-5
 - state encodings 8-2, 8-3, 8-4
 - state vector 8-2, 8-3
 - synthesizing 8-7
 - verifying 8-13
- fix_multiple_port_nets attribute 3-21
- flatten hierarchy 3-13
- flip-flops
 - negative edge 9-13
- Formality
 - default.svf 13-2
 - set_svf command 13-2
- FSM
 - finite state machine 8-1
 - optimization 1-11
- fsm_auto_inferring variable 8-9
- fsm_enable_state_minimization variable 8-9, 8-13
- fsm_export_formality_state_info variable 8-9
- full compile 1-9

G

- gated clock inference example 14-15
- gate-level netlist
 - preventing assign statements 3-21
- gray encoding style for FSMs 8-5

group command 3-6
 group -fsm command 8-7
 group_path command 11-4
 -critical_range option 3-5

H

hdlin_infer_mux variable 6-6
 hierarchical compile 2-10
 hierarchical design
 optimization strategy 2-12
 hierarchy
 flatten 3-13
 remove 3-13
 high-effort compile 3-7

I

implementation selection, synthetic library
 1-12
 incremental compile 3-8
 incremental mapping 2-6
 input port
 isolation
 propagating constraints 3-11
 set_isolate_ports command 3-9
 size_only attribute 3-10
 supporting commands 3-12

L

latch-based designs
 code examples
 D latch with asynchronous reset, Verilog 14-7
 D latch with asynchronous reset, VHDL 14-7
 D latch with asynchronous set and reset, Verilog 14-11
 D latch with asynchronous set and reset, VHDL 14-10

D latch with enable and asynchronous reset, Verilog 14-19
 D latch with enable and asynchronous reset, VHDL 14-18
 D latch with enable and asynchronous set and reset, Verilog 14-25
 D latch with enable and asynchronous set and reset, VHDL 14-24
 D latch with enable and asynchronous set, Verilog 14-22
 D latch with enable and asynchronous set, VHDL 14-21
 D latch with enable, Verilog 14-14
 D latch with enable, VHDL 14-13
 D latch, VHDL 14-5
 SR latch, Verilog 14-3
 SR latch, VHDL 14-2

libraries

multiple, using for min/max delay analyses 3-20
 optimization and 1-4
 vendor
 multibit cells in 7-1

M

mapping
 incremental 2-6
 mapping optimization
 control 2-5
 effort levels 2-5
 maximum delay, calculating cost 11-4
 Mealy finite state machine 8-3
 minimum and maximum optimization 3-18
 minimum and maximum timing analysis 3-18
 minimum delay, calculating cost 11-6
 Moore finite state machine 8-3
 multibit components
 creating 7-4
 removing 7-4, 7-6
 reporting 7-1, 7-5
 multibit library cells 7-1

multibit optimization

library cell requirements 7-6

multibit_width attribute 7-7

multiple clocks per register

optimization 3-14

setting network latency 3-15

multiplexer

library cell requirements 6-7

mapping on asynchronous signal lines 6-7

MUX_OP cell 6-4

naming convention 6-5

multiplexing logic

Design Compiler implementation 6-4

implement conditional operations implied by
if and case statements 6-3

MUX_OP cells 6-1

preferentially map multiplexing logic to
multiplexers 6-1

SELECT_OP cells 6-1

with if and case statements 6-1

MUX_OP cell 6-4

naming convention 6-5

N

negative edge flip-flops 9-13

negative slack

ignored 3-8

netlist SCANDEF flow 12-5

nets

buffering 3-21

connected to multiple ports 3-21

feedthrough, eliminating 3-21

heavily loaded, fixing 3-6

O

one-hot encoding style for FSMs 8-5

operating conditions

optimization 3-18

optimization

all paths 3-8

area 3-4

constraints 11-4

delay 3-3

flow 1-9

flow chart 1-10

FSM 1-11

hierarchical compile 2-10

how it works 11-10

incremental 3-8

mapping effort levels 2-5

mapping sequential cells 1-7

minimum and maximum 3-18

multiple clocks per register 3-14

operating conditions 3-18

process 1-9

technology libraries and 1-4

technology-specific 1-7

timing-critical sequential cells 1-8

trials phase 11-10

optimize_registers

command syntax 1-13

output port

isolation

propagating constraints 3-11

set_isolate_ports command 3-9

size_only attribute 3-10

supporting commands 3-12

P

path groups

and delay cost 11-5

creating 11-4

defined 11-4

listing 11-5

port

isolation, input and output 3-9

R

- read_scan_def 12-6
- read_scan_def consistency report 12-6
- register
 - report the type 9-7
 - type
 - report for cell 9-7
 - report for design 9-7
- register implementations 9-6
- remove_isolate_ports command 3-11
- remove_multibit command 7-4, 7-6
- report_cell command 9-7
- report_design command 3-19, 9-7
- report_design command and output example 9-7
- report_fsm command 8-7, 8-13
- report_isolate_ports command 3-11
- report_multibit command 7-5
- report_path_group command 11-5
- report_port command 3-19
- report_timing command 3-12
- report_transitive_fanout command 3-16
- report_wire_load command 3-19
- reports
 - examples 9-7
 - finite state machine 8-13
 - multibit component 7-1
 - register types
 - for design 9-7
- resource sharing 5-6
- resource sharing, enhanced 5-9
- resynthesis
 - critical path 3-12
- retiming
 - forward example 9-32
- routing congestion 7-7

S

- sample scripts 2-11
- scan chains
 - generating a SCANDEF file 12-9
 - identifying 12-6
 - importing and exporting 12-5
 - netlist SCANDEF flow 12-8
 - read_scan_def consistency report 12-6
 - reading a SCANDEF file 12-6
- scan style
 - selecting 9-24
 - specifying 9-24
- scripts
 - compile strategy 2-11
- SEQGEN
 - definition 1-12
- sequential cells
 - eliminating constant output cells 9-9
 - optimization mapping 1-7
 - removing unconnected cells 9-9
 - timing critical optimization 1-8
- set_case_analysis command 3-14
- set_clock_latency command 3-15
- set_cost_priority command 11-8
- set_critical_range command 3-5
- set_disable_timing command 3-14
- set_dont_touch_network command 3-16
- set_dont_use command 7-7
- set_fix_multiple_port_nets command 3-21
- set_fsm_encoding command 8-4, 8-7, 8-13
- set_fsm_encoding_style command 8-5, 8-7, 8-13
- set_fsm_minimize command 8-7
- set_fsm_order command 8-7
- set_fsm_preserve_state command 8-7
- set_fsm_state_vector command 8-4, 8-7, 8-13
- set_isolate_ports command 3-9
- set_max_area command
 - ignore_tns option 3-8
- set_min_library command 3-20

- set_operating_conditions command 3-20
- set_register_type command 9-6
- set_scan_configuration command
 - style option 9-20, 9-21, 9-24
- set_svf 13-2
- set_ungroup command 3-13
- set_vsdc command 13-5
- slack
 - critical negative 11-5
 - total negative 11-6
 - worst negative 11-4
- slave clock pins
 - setting default signal 3-24
- SolvNet
 - accessing xix
 - documentation xvi
- specifying
 - libraries
 - DesignWare 5-5
 - scan style 9-24
- SR latch example 14-2
- state assignment for FSMs 8-2
- state encoding styles for FSMs 8-3, 8-5
- state encodings for FSMs 8-2, 8-3, 8-4
- state vector of an FSM 8-2, 8-3
- structured logic
 - synthesis of 7-1
- style option, set_scan_configuration
 - command 9-20, 9-21, 9-24
- supported commands
 - automatic compile flow for FSMs 8-7
- synthesis of data path logic 7-1
- synthesis of structured logic 7-1
- synthesizing FSMs 8-7
- synthetic library, implementation selection 1-12

T

- technology-specific optimization 1-7

- test_default_scan_style variable 9-20, 9-21, 9-24
- test-ready compile 9-24
- timing
 - exploring design space 1-3
- timing analysis
 - minimum and maximum 3-18
- timing_enable_multiple_clocks_per_reg
 - variable 3-14
- top-down hierarchical compile strategy 2-10
- total negative slack, defined 11-6
- transition time
 - cost calculation 11-3

U

- ungroup command 3-13

V

- variables
 - auto_ungroup_preserve_constraints 4-9
 - bus_multiple_separator_style 7-5
 - bus_range_separator_style 7-5
 - compile_auto_ungroup_override_wlm 4-10
 - compile_delete_unloaded_sequential_cells 9-9
 - compile_enable_register_merging variable 9-12
 - compile_log_format 11-10
 - compile_seqmap_enable_output_inversion 9-12
 - compile_seqmap_propagate_constants 9-10
 - compile_seqmap_propagate_high_effort 11-12
 - compile_top_all_paths 2-9
 - for compile 2-5
 - fsm_auto_inferring 8-9
 - fsm_enable_state_minimization 8-9, 8-13
 - fsm_export_formality_state_info 8-9
 - hdlin_infer_mux 6-6

- test_default_scan_style 9-20, 9-21, 9-24
- timing_enable_multiple_clocks_per_reg
variable 3-14

verification

- adjusting optimization 13-3
- using Formality 13-2
- using the set_verification_priority command
13-3

- using third-party tools 13-5
 - set_vsdcc command 13-5
- verifying FSMs 8-13

W

- worst negative slack, defined 11-4
- write_scan_def 12-9