

Design Compiler[®]

User Guide

Version W-2004.12, December 2004

Comments?

Send comments on the documentation by going to <http://solvnnet.synopsys.com>, then clicking "Enter a Call to the Support Center."

SYNOPSYS[®]

Copyright Notice and Proprietary Information

Copyright © 2005 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.

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

"This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____."

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.

Registered Trademarks (®)

Synopsys, AMPS, Arcadia, C Level Design, C2HDL, C2V, C2VHDL, Cadabra, Calaveras Algorithm, CATS, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSPICE, Hypermodel, I, iN-Phase, in-Sync, Leda, MAST, Meta, Meta-Software, ModelAccess, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PowerMill, PrimeTime, RailMill, Raphael, RapidScript, Saber, SiVL, SNUG, SolvNet, Stream Driven Simulator, Superlog, System Compiler, Testify, TetraMAX, TimeMill, TMA, VCS, Vera, and Virtual Stepper are registered trademarks of Synopsys, Inc.

Trademarks (™)

abraCAD, abraMAP, Active Parasitics, AFGen, Apollo, Apollo II, Apollo-DPII, Apollo-GA, ApolloGAI, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanTestchip, AvanWaves, BCView, Behavioral Compiler, BOA, BRT, Cedar, ChipPlanner, Circuit Analysis, Columbia, Columbia-CE, Comet 3D, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, Cyclelink, Davinci, DC Expert, DC Expert *Plus*, DC Professional, DC Ultra, DC Ultra Plus, Design Advisor, Design Analyzer, Design Vision, DesignerHDL, DesignTime, DFM-Workbench, DFT Compiler, Direct RTL, Direct Silicon Access, Discovery, DW8051, DWPCI, Dynamic Model Switcher, Dynamic-Macromodeling, ECL Compiler, ECO Compiler, EDAnavigator, Encore, Encore PQ, Evaccess, ExpressModel, Floorplan Manager, Formal Model Checker, FoundryModel, FPGA Compiler II, FPGA *Express*, Frame Compiler, Galaxy, Gatran, HDL Advisor, HDL Compiler, Hercules, Hercules-Explorer, Hercules-II, Hierarchical Optimization Technology, High Performance Option, HotPlace, HSPICE-Link, i-Virtual Stepper, iN-Tandem, Integrator, Interactive Waveform Viewer, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, JvXtreme, Liberty, Libra-Passport, Libra-Visa, Library Compiler, Magellan, Mars, Mars-Rail, Mars-Xtalk, Medici, Metacapture, Metacircuit, Metamanager, Metamixsim, Milkyway, ModelSource, Module Compiler, MS-3200, MS-3400, Nova Product Family, Nova-ExploreRTL, Nova-Trans, Nova-VeriLint, Nova-VHDLint, Optimum Silicon, Orion_ec, Parasitic View, Passport, Planet, Planet-PL, Planet-RTL, Polaris, Polaris-CBS, Polaris-MT, Power Compiler, PowerCODE, PowerGate, ProFPGA, ProGen, Prospector, Protocol Compiler, PSMGen, Raphael-NES, RoadRunner, RTL Analyzer, Saturn, ScanBand, Schematic Compiler, Scirocco, Scirocco-i, Shadow Debugger, Silicon Blueprint, Silicon Early Access, SinglePass-SoC, Smart Extraction, SmartLicense, SmartModel Library, Softwire, Source-Level Design, Star, Star-DC, Star-MS, Star-MTB, Star-Power, Star-Rail, Star-RC, Star-RCXT, Star-Sim, Star-SimXT, Star-Time, Star-XP, SWIFT, Taurus, Taurus-Device, Taurus-Layout, Taurus-Lithography, Taurus-Process, Taurus-Topography, Taurus-Visual, Taurus-Workbench, TimeSlice, TimeTracker, Timing Annotator, TopoPlace, TopoRoute, Trace-On-Demand, True-Hspice, TSUPREM-4, TymeWare, VCS Express, VCSi, Venus, Verification Portal, VFormal, VHDL Compiler, VHDL System Simulator, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (SM)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.

ARM and AMBA are registered trademarks of ARM Limited.

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

Printed in the U.S.A.

Document Order Number: 36042-000 WB
Design Compiler User Guide, version W-2004.12

Contents

What's New in This Release	xviii
About This Manual	xxv
Customer Support	xxix
1. Introduction to Design Compiler	
Design Compiler and the Design Flow	1-2
Design Compiler Family	1-4
DC Expert	1-5
DC Ultra	1-6
HDL Compiler Tools	1-6
DesignWare Library	1-7
DFT Compiler	1-7
Module Compiler	1-7
Power Compiler	1-7
Design Vision	1-8
Design Compiler FPGA	1-8

2. Design Compiler Basics

The High-Level Design Flow	2-3
Running Design Compiler	2-6
Design Compiler Modes.	2-7
Design Compiler Interfaces	2-7
Setup Files.	2-8
Starting Design Compiler.	2-10
Exiting Design Compiler.	2-11
Getting Command Help	2-12
Using Command Log Files.	2-13
Using the Filename Log File	2-14
Using Script Files.	2-14
Working with Licenses	2-15
Listing the Licenses in Use.	2-15
Getting Licenses.	2-16
Releasing Licenses	2-16
Following the Basic Synthesis Flow	2-16
A Design Compiler Session Example	2-24

3. Preparing Design Files for Synthesis

Managing the Design Data	3-2
Controlling the Design Data.	3-2
Organizing the Design Data.	3-3
Partitioning for Synthesis.	3-4
Partitioning for Design Reuse	3-5

Keeping Related Combinational Logic Together	3-5
Registering Block Outputs	3-7
Partitioning by Design Goal	3-8
Partitioning by Compile Technique.	3-9
Keeping Sharable Resources Together	3-10
Keeping User-Defined Resources With the Logic They Drive . .	3-11
Isolating Special Functions	3-12
HDL Coding for Synthesis	3-13
Writing Technology-Independent HDL	3-14
Inferring Components.	3-14
Using Synthetic Libraries	3-18
Designing State Machines	3-21
Using HDL Constructs	3-22
General HDL Constructs	3-22
Using Verilog Macro Definitions	3-27
Using VHDL Port Definitions	3-28
Writing Effective Code	3-28
Guidelines for Identifiers.	3-28
Guidelines for Expressions.	3-30
Guidelines for Functions.	3-31
Guidelines for Modules.	3-33
 4. Working With Libraries	
Selecting a Semiconductor Vendor	4-3
Understanding the Library Requirements	4-3
Technology Libraries	4-4
Symbol Libraries	4-6

DesignWare Libraries	4-6
Specifying Libraries	4-7
Specifying a Library Search Path.	4-7
Specifying Technology Libraries	4-8
Specifying DesignWare Libraries.	4-10
Loading Libraries.	4-11
Listing Libraries	4-11
Reporting Library Contents	4-12
Specifying Library Objects.	4-12
Directing Library Cell Usage	4-13
Excluding Cells From the Target Library	4-13
Specifying Cell Preferences.	4-14
Removing Libraries From Memory	4-16
Saving Libraries.	4-16
 5. Working With Designs in Memory	
Design Terminology.	5-3
About Designs	5-3
Flat Designs	5-3
Hierarchical Designs	5-3
Design Objects.	5-4
Current Design	5-5
Cells, Instances, and References.	5-5
Nets	5-7
Ports.	5-7

Pins	5-7
Reading Designs	5-7
Using a Search Path	5-9
Reading .db Files	5-10
Reading HDL Designs	5-10
Analyzing Designs	5-11
Elaborating Designs	5-11
Listing Designs in Memory	5-11
Setting the Current Design	5-13
Linking Designs	5-14
Linking a Design Manually	5-15
Linking a Design Automatically	5-16
Changing Design References	5-16
Listing Design Objects	5-17
Specifying Design Objects	5-18
Using a Relative Path	5-18
Using an Absolute Path	5-20
Creating Designs	5-21
Copying Designs	5-22
Renaming Designs	5-23
Changing the Design Hierarchy	5-24
Adding Levels of Hierarchy	5-24
Grouping Cells Into Subdesigns	5-25
Grouping Related Components Into Subdesigns	5-26

Removing Levels of Hierarchy	5-27
Ungrouping Hierarchies Before Optimization	5-28
Ungrouping Hierarchies During Optimization	5-30
Cell Count-Based Auto-Ungrouping	5-32
Delay-Based Auto-Ungrouping	5-33
Cases in Which Auto-Ungrouping Is Not Performed	5-33
Preserving Hierarchical Pin Timing Constraints During Ungrouping	5-35
Merging Cells From Different Subdesigns	5-37
Editing Designs	5-38
Translating Designs From One Technology to Another	5-40
Procedure to Translate Designs	5-40
Restrictions on Translating Between Technologies	5-41
Removing Designs From Memory	5-42
Saving Designs	5-43
Saving Designs To Other Formats	5-44
Ensuring Name Consistency Between the Design Database and the Netlist	5-45
Working With Attributes	5-50
Setting Attribute Values	5-51
Viewing Attribute Values	5-52
Saving Attribute Values	5-53
Defining Attributes	5-53
Removing Attributes	5-54
The Object Search Order	5-55

6. Defining the Design Environment

Defining the Operating Conditions	6-3
Determining Available Operating Condition Options	6-4
Specifying Operating Conditions	6-5
Defining Wire Load Models	6-5
Hierarchical Wire Load Models	6-7
Determining Available Wire Load Models	6-9
Specifying Wire Load Models and Modes	6-11
Modeling the System Interface	6-13
Defining Drive Characteristics for Input Ports	6-13
The set_driving_cell Command	6-14
The set_drive and set_input_transition Commands	6-15
Defining Loads on Input and Output Ports	6-17
Defining Fanout Loads on Output Ports	6-18

7. Defining Design Constraints

Setting Design Rule Constraints	7-3
Setting Transition Time Constraints	7-4
Setting Fanout Load Constraints	7-5
Setting Capacitance Constraints	7-7
Setting Optimization Constraints	7-9
Setting Timing Constraints	7-9
Defining a Clock	7-11
Specifying I/O Timing Requirements	7-13
Specifying Combinational Path Delay Requirements	7-16
Specifying Timing Exceptions	7-16

Setting Area Constraints	7-25
Verifying the Precompiled Design	7-26
8. Optimizing the Design	
The Optimization Process	8-2
Architectural Optimization	8-2
Logic-Level Optimization	8-3
Gate-Level Optimization.	8-6
Selecting and Using a Compile Strategy.	8-7
Top-Down Compile.	8-9
Bottom-Up Compile	8-12
Mixed Compile Strategy.	8-19
Resolving Multiple Instances of a Design Reference	8-20
Uniquify Method.	8-22
Compile-Once-Don't-Touch Method.	8-24
Ungroup Method	8-26
Preserving Subdesigns	8-28
Understanding the Compile Cost Function	8-30
Calculating Transition Time Cost	8-31
Calculating Fanout Cost.	8-32
Calculating Capacitance Cost	8-32
Calculating Cell Degradation Cost.	8-33
Calculating Maximum Delay Cost	8-33
Worst Negative Slack Method	8-33
Critical Range Negative Slack Method.	8-35

Calculating Minimum Delay Cost	8-36
Calculating Maximum Power Cost	8-37
Calculating Maximum Area Cost	8-38
Calculating Minimum Porosity Cost	8-38
Performing Design Exploration	8-39
Performing Design Implementation	8-40
Optimizing Random Logic	8-40
Optimizing Structured Logic.	8-42
Optimizing High-Speed Designs	8-42
Optimizing for Maximum Performance.	8-44
Creating Path Groups.	8-45
Fixing Heavily Loaded Nets	8-48
Flattening Logic on the Critical Path.	8-49
Automatically Ungrouping Hierarchies on the Critical Path .	8-51
Performing a High-Effort Compile.	8-51
Performing a High-Effort Incremental Compile.	8-52
Optimizing for Minimum Area.	8-53
Disabling Total Negative Slack Optimization	8-53
Enabling Sequential Area Recovery.	8-54
Enabling Boolean Optimization	8-54
Managing Resource Selection	8-55
Using Flattening	8-56
Optimizing Across Hierarchical Boundaries	8-56
Optimizing Data Paths	8-58
Using DC Ultra Datapath Optimization	8-58
Datapath Extraction	8-60
Two Different Datapath Optimization Methods.	8-64

Methodology Flow	8-67
Datapath Report	8-70
Commands Specific to DC Ultra Datapath Optimization	8-73
 9. Analyzing and Resolving Design Problems	
Checking for Design Consistency	9-2
Analyzing Your Design During Optimization	9-3
Customizing the Compile Log	9-3
Saving Intermediate Design Databases.	9-5
Manual Checkpointing	9-6
Automatic Checkpointing	9-7
Analyzing Design Problems.	9-8
Analyzing Timing Problems.	9-9
Resolving Specific Problems.	9-10
Analyzing Cell Delays	9-11
Finding Unmapped Cells	9-13
Finding Black Box Cells	9-14
Finding Hierarchical Cells	9-14
Disabling Reporting of Scan Chain Violations	9-14
Insulating Interblock Loading	9-16
Preserving Dangling Logic.	9-16
Preventing Wire Delays on Ports	9-16
Breaking a Feedback Loop	9-17
Analyzing Buffer Problems.	9-17
Understanding Buffer Insertion.	9-17

Correcting for Missing Buffers	9-23
Correcting for Extra Buffers	9-26
Correcting for Hanging Buffers	9-27
Correcting Modified Buffer Networks	9-27
10. Creating, Instantiating, and Using Interface Logic Models in Hierarchical Synthesis	
Introduction to Interface Logic Models.	10-3
Benefits of Interface Logic Models	10-5
Considerations When Using Interface Logic Models.	10-6
Interface Logic Model Handling by Design Compiler.	10-8
Guidelines for Creating an Interface Logic Model	10-9
Creating Interface Logic Models	10-11
Controlling the Logic Included in an Interface Logic Model.	10-14
Controlling the Fanins and Fanouts of Chip-level Networks . . .	10-15
Controlling the Number of Latch Levels.	10-16
Controlling Side-load Cells.	10-18
Using the create_ilm Command Options.	10-19
Saving Interface Logic Models	10-21
Note on Changed Names for Common Subdesigns	10-22
Sample Scripts to Create an Interface Logic Model	10-24
Reporting Information About Interface Logic Models	10-27
Reporting Design Information	10-28
Reporting Area Information	10-29

Instantiating and Using Interface Logic Models in Logical Synthesis	10-30
Sample Script to Instantiate Interface Logic Models and Run Logical Synthesis in a Hierarchical Route Flow . . .	10-32
Back-Annotation With Interface Logic Models	10-33
Applying Back-Annotation Data for Interface Logic Models	10-34
Applying Back-Annotated Data on the Original Block	10-35
Applying Back-Annotation Data on the Interface Logic Model Set as the Current Design	10-35
Applying Back-Annotation Data on the Interface Logic Model Subdesign With the Current Design as the Top-Level Design.	10-36
Handling Interface Logic Model Nets That Have Back-Annotated Data	10-36
Handling Interface Logic Model Cells That Have Back-Annotated Data	10-37
Summary of Commands for Interface Logic Models	10-37
Appendix A. Design Example	
Design Description	A-2
Setup File	A-12
Default Constraints File	A-13
Compile Scripts	A-16
Appendix B. Basic Commands	
Commands for Defining Design Rules	B-2
Commands for Defining Design Environments	B-2

Commands for Setting Design Constraints B-3

Commands for Analyzing and Resolving Design Problems B-5

Appendix C. Predefined Attributes

Glossary

Index

Preface

This preface includes the following sections:

- [What's New in This Release](#)
- [About This Manual](#)
- [Customer Support](#)

What's New in This Release

This section describes the new features, enhancements, and changes included in Design Compiler version W-2004.12. Unless otherwise noted, you can find additional information about these changes later in this book.

Design Compiler version W-2004.12 yields significant improvements in runtime and quality of results (QoR), compared to version V-2004.06. Specific enhancements are as follows:

- Quality of Results (QoR)
 - Automatic Removal of Constant Registers

In version W-2004.12, the `compile_seqmap_propagate_constants` variable is true by default, thereby enabling automatic removal of constant registers. For more information, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*, Chapter 3.

- Register Mapping

Design Compiler version W-2004.12 delivers a new register mapping engine, which provides better register mapping. The new mapping engine is on by default. With the improved initial register mapper, you might not need to use sequential area recovery (enabled with `set compile_sequential_area_recovery true`) to improve results. For more information on area recovery, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*, Chapter 3.

- Support of Automatic Boundary Optimization During the `compile_ultra` Flow

In version W-2004.12, boundary optimization is turned on by default in the `compile_ultra` flow. You can disable boundary optimization by using the `-no_boundary_optimization` option of the `compile_ultra` command. With this option turned on, no hierarchical boundary optimization is performed. For more information, see Chapter 8 of this manual.

- Improvement in Critical Path Resynthesis Optimization

Design Compiler version W-2004.12 uses a new resynthesis mapper to deliver better runtime and QoR. This version has improved library support and correctly handles XORs and MUXes. For more information on critical path resynthesis, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*, Chapter 1.

- New Datapath Generator

In previous versions, Design Compiler supported the MC-Inside-DW flow, which used Module Compiler to generate optimal results for datapath designs. In version W-2004.12 a new datapath generator delivers equal or better QoR without using Module Compiler. The new datapath generator is on by default with the following settings:

```
set synthetic_library {dw_foundation.sldb}  
set synlib_enable_dpgen true
```

With the `synlib_enable_dpgen` variable set to true by default, Design Compiler ignores the setting of the `dw_prefer_mc_inside` variable. For more information, see Chapter 8 of this manual.

- Datapath Extraction Report

In previous versions, when you enabled DC Ultra datapath optimization by setting the `set_ultra_optimization` command to true or running the `compile_ultra` command, DC Ultra performed datapath extraction after exploring various datapath and resource-sharing options during compile.

Design Compiler version W-2004.12 generates a detailed datapath extraction report after compile. An enhanced `report_resources` command displays detailed information on how the datapath blocks were extracted. In addition, during compile, the datapath extraction report prints out datapaths from earlier phases of compile if the `compile_report_dp` variable is set to true. For more information, see Chapter 8 of this manual.

- DesignWare Foundation Library Usage Model for DC Ultra Optimization Flow

In previous versions, you explicitly added the `dw_foundation.sldb` to the synthetic library in order to utilize the QoR benefits provided by the licensed DesignWare implementations for singleton DesignWare parts. In Design Compiler version W-2004.12, by default, `dw_foundation.sldb` is added automatically to the synthetic library when you use the `compile_ultra` or `set_ultra_optimization` command. If you do not want Design Compiler to add the `dw_foundation.sldb` library to the synthetic library, use the `-no_auto_dwlib` option of the `set_ultra_optimization` command. For more information, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*, Chapter 1.

- Timing
 - I/O Latency Estimation

In Design Compiler version W-2004.12, you can have the tool estimate the I/O latency (network latency at input ports and output ports) in the current design clock tree and apply it to the input delay and output delay. To do so, you set the `estimate_io_latency` variable to true. For more information, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*, Chapter 12.

- Conditional Arc Support in Standard Delay Format (SDF)

In previous versions, Design Compiler supported conditional timing arc analysis only from library cells, with “when” constructs in the library.

Design Compiler version W-2004.12 extends this support to the SDF flow. The tool can read in SDF files and back-annotate conditional arc delays for analysis based on constant values of cell inputs, which results in more accurate timing analysis. You can back-annotate conditional arc information from PrimeTime, Astro, or any other back-end tool. For more information, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*, Chapter 8.

- Ideal Clock Network Latency per pin per Clock

Design Compiler version W-2004.12 allows you to set clock network latency on a pin or port with reference to a specific clock. You can define a network latency for each clock signal that passes through a specific pin or port to a set of fanout registers. To do so, use the `-clock` option with the `set_clock_latency` command. This option is useful when multiple clocks might traverse the same pin or port—that is, in designs in which multiple clocks propagate to a register. For more information, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*, Chapter 3.

- Usability

- Ideal Network Enhancements

Design Compiler version W-2004.12 unifies the `set_ideal_net` and `set_ideal_network` concepts. The `set_ideal_network` command has a new option, `-no_propagate`. Starting with this release, the `set_ideal_net` command automatically defaults to the `set_ideal_network -no_propagate` command. When you specify a net as the source of the ideal network, the ideal network attributes are applied to the global driver pins or ports of the specified net. This ensures that the ideal network attributes are not lost even if the net is optimized away. For more information, see the *Design Compiler Reference Manual: Constraints and Timing*, Chapter 2.

- Support for Significant Digits Global Variable

In Design Compiler version W-2004.12, for those commands that support the `-significant_digits` option, you can set the default number of significant digits with the `report_default_significant_digits` variable. For more information, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*, Chapter 10.

- Enhanced Command and File Logging

In version W-2004.12, you can start Design Compiler with the `-no_log` option. When you use this option, Design Compiler does not log commands in the `command.log` file. In addition, Design Compiler appends the process ID of the application and date stamp to the filename log file. These enhancements allow you to run multiple instances of Design Compiler in the

same working directory without encountering any file creation conflicts. For more information, see the *Design Compiler Command-Line Interface Guide*, Chapter 1.

- User Interface Enhancements

Design Compiler version W-2004.12 supports the `-leaf` option for the `all_connected` command. When you use this option, Design Compiler returns a list of global pins of the net. In `dctcl` mode, a collection is returned. In addition, this version supports the `-noleaf` option for the `report_hierarchy` command. When you use this option, leaf library cells are excluded from the reference hierarchy report.

- Interface Logic Models (ILM)

- The `create_ilm` command has two new options, `-identify_only` and `-extract_only`. Using these options, you can customize an ILM to modify the logic included in the model.
- Beginning with this release, as part of creating an ILM, the `create_ilm` command removes the `is_interface_logic` attribute from the nets, cells, and pins comprising the model.
- In this version, you can prevent the `get_ilms` command from issuing messages by using the new `-quiet` option.

For more information on interface logic models, see Chapter 10 of this manual.

- Formality Verification

Design Compiler version W-2004.12 automatically creates a default Setup Verification File (SVF), named `default.svf`, in your working directory. You should include this file in Formality to help in compare point matching and verification. You can use the

`set_svf` command to control the name of the file or generate it in a different location. For DesignWare specific operations, a directory named `dwsvf_*` is created at the same location as the default.svf file. Formality automatically reads the contents in this directory when you issue the `set_svf` command. For more information, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*, Chapter 7, and the Formality documentation.

- Enhanced Register Retiming
 - Design Compiler version W-2004.12 introduces the `optimize_reg_always_insert_sequential` variable. When this variable is set to true, the `optimize_registers` command removes all movable sequential cells from the design and reinserts and remaps them (even if no register was moved) to potentially reduce area cost. If your design contains register trees that were not built optimally, setting this variable to true might reduce the register count. For more information, see the *Design Compiler Reference Manual: Register Retiming*.
 - Design Compiler version W-2004.12 provides support for register retiming of designs in which nets have multiple drivers and bidirectional pins. For more information on register retiming, see the *Design Compiler Reference Manual: Register Retiming*.

Known Limitations and Resolved STARs

Information about known problems and limitations, as well as about 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 Synopsys Web page at <http://www.synopsys.com> and click SolvNet.
2. 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.)
3. Click Release Notes in the Main Navigation section (on the left), move your pointer to Design Compiler, then choose W-2004.12 in the menu that appears.

The W-2004.12 Release Notes appear.

- For major releases, click Design Compiler in the Product Release Notes section (on the left) to open the *Design Compiler Release Notes*.
- For bug fix or service pack releases, separate product-specific release notes are not provided.

About This Manual

The *Design Compiler User Guide* provides basic synthesis information for users of the Design Compiler tools. This manual describes synthesis concepts and commands, and presents examples for basic synthesis strategies.

This manual does not cover asynchronous design, I/O pad synthesis, test synthesis, simulation, physical design techniques (such as floorplanning or place and route), or back-annotation of physical design information.

The information presented here supplements the Synopsys synthesis reference manuals but does not replace them. See other Synopsys documentation for details about topics not covered in this manual.

This manual supports version W-2004.12 of the Synopsys synthesis tools, whether they are running under the UNIX operating system or the Linux operating system. The main text of this manual describes UNIX operation.

Unless otherwise specified, all Design Compiler shell commands presented in this manual work in both `dcsh` and `dctcl` command-line modes. When the command syntax is the same for both `dcsh` and `dctcl` command-line modes, the manual provides a single command example, preceded with the `dc_shell>` prompt. When the command syntax differs between `dcsh` and `dctcl` command-line modes, the manual provides two command examples. In this case, the `dcsh` example is preceded by the `dc_shell>` prompt, while the `dctcl` example is preceded by the `dc_shell-t>` prompt.

Audience

This manual is intended for logic designers and engineers who use the Synopsys synthesis tools with the VHDL or Verilog hardware description language (HDL). Before using this manual, you should be familiar with the following topics:

- High-level design techniques
- ASIC design principles
- Timing analysis principles
- Functional partitioning techniques

Related Publications

For additional information about Design Compiler, see

- Synopsys Online Documentation (SOLD), which is included with the software for CD users or is available to download through the Synopsys electronic software transfer (EST) system
- Documentation on the Web, which is available through SolvNet at <http://solvnet.synopsys.com>
- The Synopsys MediaDocs Shop, from which you can order printed copies of Synopsys documents, at <http://mediadocs.synopsys.com>

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

- Automated Chip Synthesis
- Design Budgeting
- Design Vision
- DesignWare components
- DFT Compiler
- Floorplan Manager
- Design Compiler FPGA
- Module Compiler
- PrimeTime
- Power Compiler
- HDL Compiler

Also see the following related documents:

- *Using Tcl With Synopsys Tools*
- *Synthesis Master Index*

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
Courier	Indicates command syntax.
<i>Courier italic</i>	Indicates a user-defined value in Synopsys syntax, such as <i>object_name</i> . (A user-defined value that is not Synopsys syntax, such as a user-defined value in a Verilog or VHDL statement, is indicated by regular text font italic.)
Courier bold	Indicates user input—text you type verbatim—in Synopsys syntax and examples. (User input that is not Synopsys syntax, such as a user name or password you enter in a GUI, is indicated by regular text font bold.)
[]	Denotes optional parameters, such as <i>pin1 [pin2 ... pinN]</i>
	Indicates a choice among alternatives, such as <i>low medium high</i> (This example indicates that you can enter one of three possible values for an option: low, medium, or high.)
—	Connects terms that are read as a single term by the system, such as <i>set_annotated_delay</i>
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.

Convention	Description
\	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 an electronic 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 on the Web, and “Enter a Call to the Support Center.”

To access SolvNet,

1. Go to the SolvNet Web page at <http://solvnet.synopsys.com>.
2. 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 SolvNet Help in the Support Resources section.

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 call to your local support center from the Web by going to <http://solvnet.synopsys.com> (Synopsys user name and password required), then clicking “Enter a Call to the Support Center.”
- 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/support_ctr.
- Telephone your local support center.
 - Call (800) 245-8005 from within the continental United States.
 - Call (650) 584-4200 from Canada.
 - Find other local support center telephone numbers at http://www.synopsys.com/support/support_ctr.

1

Introduction to Design Compiler

The Design Compiler tool is the core of the Synopsys synthesis products. Design Compiler optimizes designs to provide the smallest and fastest logical representation of a given function. It comprises tools that synthesize your HDL designs into optimized technology-dependent, gate-level designs. It supports a wide range of flat and hierarchical design styles and can optimize both combinational and sequential designs for speed, area, and power.

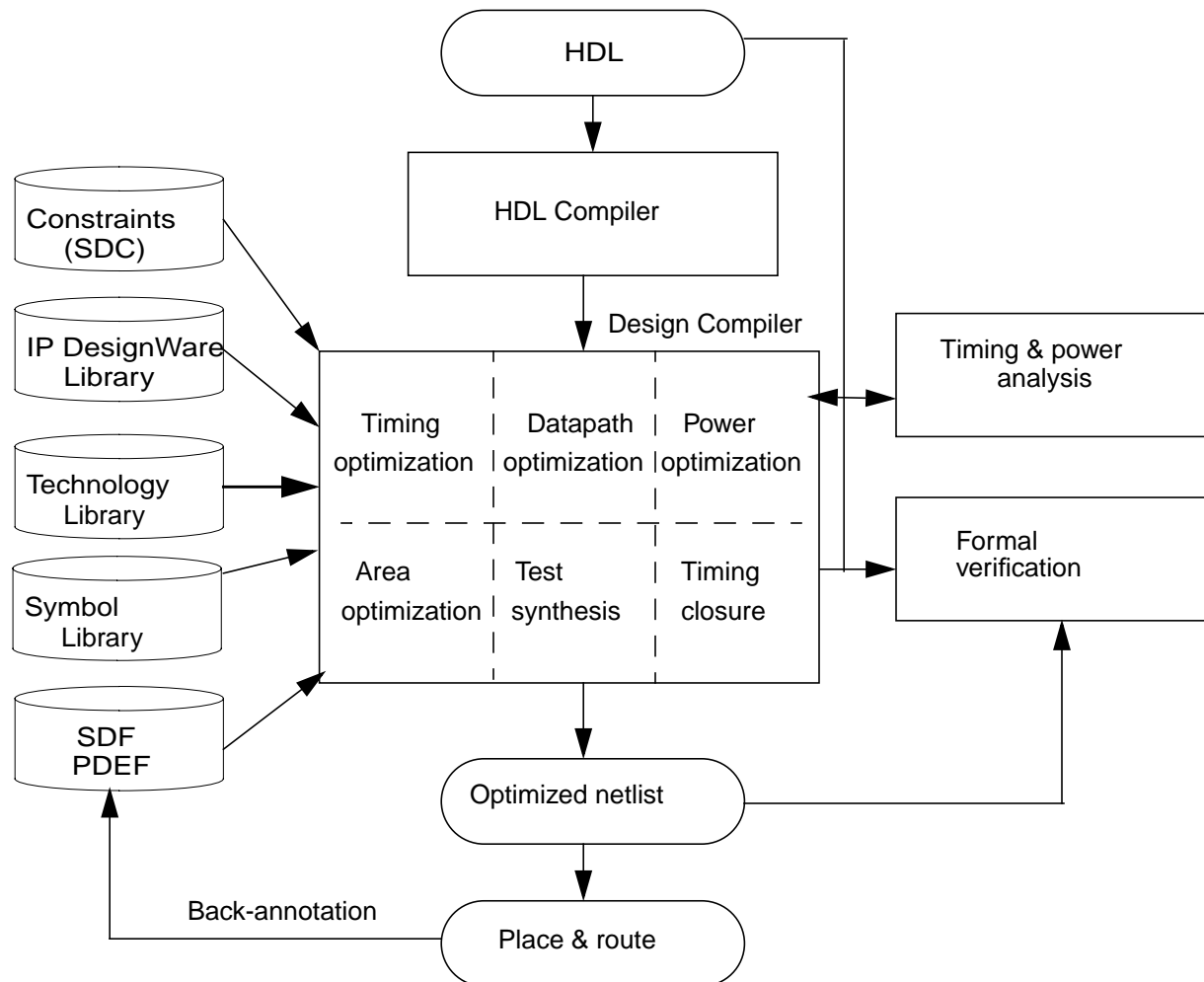
This chapter includes the following sections:

- [Design Compiler and the Design Flow](#)
- [Design Compiler Family](#)

Design Compiler and the Design Flow

Figure 1-1 shows a simplified overview of how Design Compiler fits into the design flow.

Figure 1-1 Design Compiler and the Design Flow



You use Design Compiler for logic synthesis, which is the process of converting a design description written in a hardware description language such as Verilog or VHDL into an optimized gate-level netlist mapped to a specific technology library. The steps in the synthesis process are as follows:

1. The input design files for Design Compiler are often written using a hardware description language (HDL) such as Verilog or VHDL.
2. Design Compiler uses technology libraries, synthetic or DesignWare libraries, and symbol libraries to implement synthesis and to display synthesis results graphically.

During the synthesis process, Design Compiler translates the HDL description to components extracted from the generic technology (GTECH) library and DesignWare library. The GTECH library consists of basic logic gates and flip-flops. The DesignWare library contains more complex cells such as adders and comparators. Both the GTECH and DesignWare libraries are technology independent, that is, they are not mapped to a specific technology library. Design Compiler uses the symbol library to generate the design schematic.

3. After translating the HDL description to gates, Design Compiler optimizes and maps the design to a specific technology library, known as the target library. The process is constraint driven. Constraints are the designer's specification of timing and environmental restrictions under which synthesis is to be performed.
4. After the design is optimized, it is ready for test synthesis. Test synthesis is the process by which designers can integrate test logic into a design during logic synthesis. Test synthesis enables designers to ensure that a design is testable and resolve any test issues early in the design cycle.

The result of the logic synthesis process is an optimized gate-level netlist, which is a list of circuit elements and their interconnections.

5. After test synthesis, the design is ready for the place and route tools, which place and interconnect cells in the design. Based on the physical routing, the designer can back-annotate the design with actual interconnect delays; Design Compiler can then resynthesize the design for more accurate timing analysis.

Design Compiler reads and writes design files in all the standard electronic design automation (EDA) formats, including Synopsys internal database (.db) and equation (.eqn) formats. In addition, Design Compiler provides links to other EDA tools, such as place and route tools, and to post-layout resynthesis techniques, such as in-place optimization. These links enable information sharing, including forward-directed constraints and delays, between Design Compiler and external tools.

Design Compiler Family

Synopsys provides an integrated RTL synthesis solution. Using Design Compiler tools, you can

- Produce fast, area-efficient ASIC designs by employing user-specified gate-array, FPGA, or standard-cell libraries
- Translate designs from one technology to another
- Explore design tradeoffs involving design constraints such as timing, area, and power under various loading, temperature, and voltage conditions

- Synthesize and optimize finite state machines, including automatic state assignment and state minimization
- Integrate netlist inputs and netlist or schematic outputs into third-party environments while still supporting delay information and place and route constraints
- Create and partition hierarchical schematics automatically

DC Expert

At the core of the Synopsys' RTL synthesis solution is the DC Expert. DC Expert is applied to high-performance ASIC and IC designs.

DC Expert provides the following features:

- Hierarchical compile (top down or bottom up)
- Full and incremental compile techniques
- Sequential optimization for complex flip-flops and latches
- Time borrowing for latch-based designs
- Timing analysis
- Buffer balancing (within hierarchical blocks)
- Command-line interface and graphical user interface
- Budgeting, the process of allocating timing and environment constraints among blocks in a design
- Automated chip synthesis, a set of Design Compiler commands that fully automate the partitioning, budgeting, and distributed synthesis flow for large designs

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 DC Expert capabilities, DC Ultra provides the following features:

- Additional high-effort delay optimization algorithms
- Advanced arithmetic optimization
- Integrated datapath partitioning and synthesis capabilities
- Finite state machine (FSM) optimization
- Advanced critical path resynthesis
- Register retiming, the process by which the tool moves registers through combinational gates to improve timing
- Support for advanced cell modeling, that is, the cell-degradation design rule
- Location-based optimization to enable faster timing closure by tightly linking the logical and physical environments
- Advanced timing analysis

HDL Compiler Tools

The HDL compiler reads HDL files and performs translation and architectural optimization of the designs. For more information about the HDL Compiler tools, see the HDL Compiler documentation.

DesignWare Library

A DesignWare library is a collection of reusable circuit-design building blocks (components) that are tightly integrated into the Synopsys synthesis environment. During synthesis, Design Compiler selects the right component with the best speed and area optimization from the DesignWare Library. For more information, see the DesignWare Library documentation.

DFT Compiler

The DFT Compiler tool is the Synopsys test synthesis solution. DFT Compiler provides integrated design-for-test capabilities, including constraint-driven scan insertion during compile. The DFT Compiler tool is applied to high-performance ASIC and IC designs that utilize scan test techniques. For more information, see the DFT Compiler documentation.

Module Compiler

The Module Compiler tool facilitates high-performance ASIC datapath design. For more information, see the *Module Compiler User Guide*.

Power Compiler

The Power Compiler tool offers a complete methodology for power, including analyzing and optimizing designs for static and dynamic power consumption. For more information about these power capabilities, see the *Power Compiler User Guide*.

Design Vision

The Design Vision is a graphical user interface (GUI) to the Synopsys synthesis environment and an analysis tool for viewing and analysing designs at the generic technology (GTECH) level and gate level. Design Vision provides menus and dialog boxes for implementing Design Compiler commands. It also provides graphical displays, such as design schematics. For more information, see the *Design Vision User Guide* and *Design Vision Help*.

Design Compiler FPGA

The Design Compiler FPGA tool enables input of FPGA technology libraries and data formats. It provides FPGA-specific optimization algorithms with features for high-performance FPGA implementations. For more information, see the *Design Compiler FPGA User Guide*.

2

Design Compiler Basics

This chapter provides basic information about Design Compiler functions. The chapter presents both high-level and basic synthesis design flows. Standard user tasks, from design preparation and library specification to compile strategies, optimization, and results analysis, are introduced as part of the basic synthesis design flow presentation.

This chapter includes the following sections:

- [The High-Level Design Flow](#)
- [Running Design Compiler](#)
- [Following the Basic Synthesis Flow](#)
- [A Design Compiler Session Example](#)

Note:

Even though the following terms have slightly different meanings, they are often used synonymously in Design Compiler documentation:

Synthesis is the process that generates a gate-level netlist for an IC design that has been defined using a Hardware Description Language (HDL). Synthesis includes reading the HDL source code and optimizing the design from that description.

Optimization is the step in the synthesis process that attempts to implement a combination of library cells that best meet the functional, timing, and area requirements of the design.

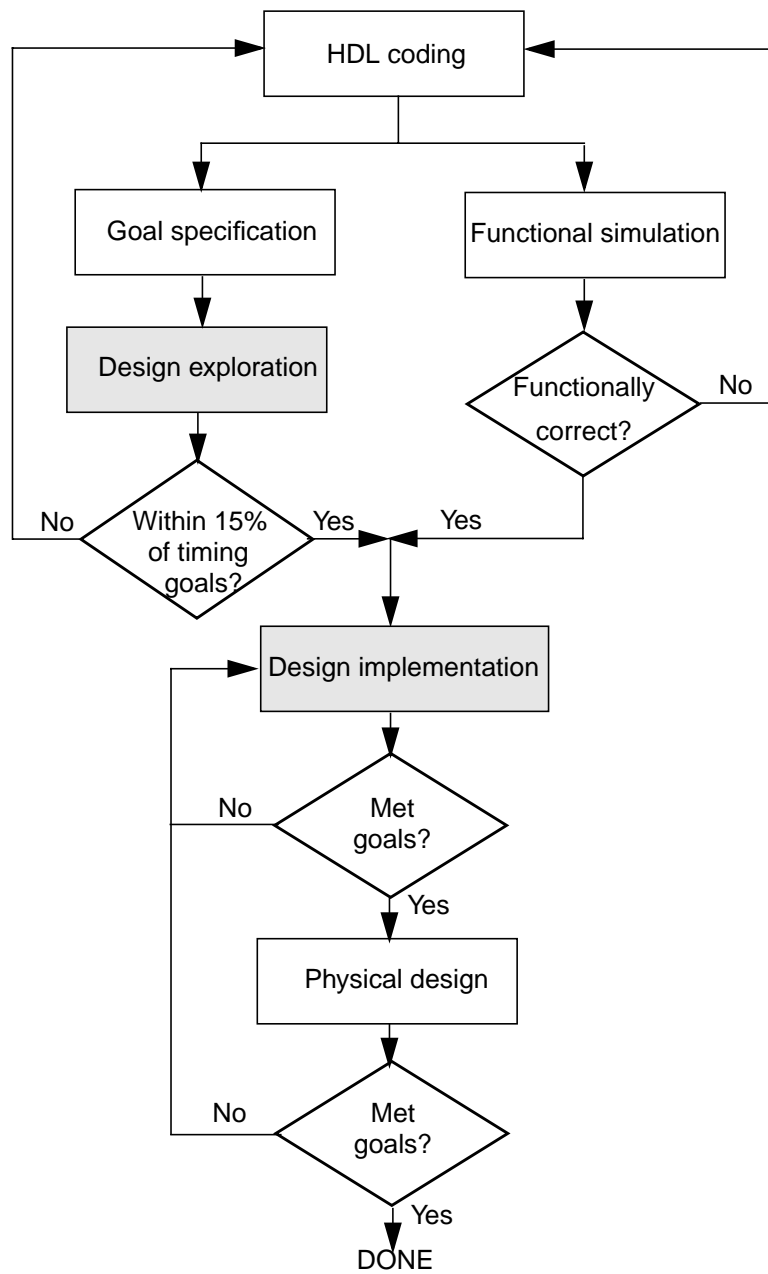
Compile is the Design Compiler command and process that executes the optimization step. After you read in the design and perform other necessary tasks, you invoke the compile command to generate a gate-level netlist for the design.

The High-Level Design Flow

In a basic high-level design flow, Design Compiler is used in both the design exploration stage and the final design implementation stage. In the exploratory stage, you use Design Compiler to carry out a preliminary, or default, synthesis. In the design implementation stage, you use the full power of Design Compiler to synthesize the design.

[Figure 2-1](#) shows the high-level design flow. The shaded areas indicate where Design Compiler synthesis tasks occur in the flow.

Figure 2-1 Basic High-Level Design Flow



Using the design flow shown in [Figure 2-1](#), you perform the following steps:

1. Start by writing an HDL description (Verilog or VHDL) of your design. Use good coding practices to facilitate successful Design Compiler synthesis of the design.
2. Perform design exploration and functional simulation in parallel.
 - In design exploration, use Design Compiler to (a) implement specific design goals (design rules and optimization constraints) and (b) carry out a preliminary, “default” synthesis (using only the Design Compiler default options).
 - If design exploration fails to meet timing goals by more than 15 percent, modify your design goals and constraints, or improve the HDL code. Then repeat both design exploration and functional simulation.
 - In functional simulation, determine whether the design performs the desired functions by using an appropriate simulation tool.
 - If the design does not function as required, you must modify the HDL code and repeat both design exploration and functional simulation.
 - Continue performing design exploration and functional simulation until the design is functioning correctly and is within 15 percent of the timing goals.
3. Perform design implementation synthesis by using Design Compiler to meet design goals.

After synthesizing the design into a gate-level netlist, verify that the design meets your goals. If the design does not meet your goals, generate and analyze various reports to determine the techniques you might use to correct the problems.

4. After the design meets functionality, timing, and other design goals, complete the physical design (either in-house or by sending it to your semiconductor vendor).

Analyze the physical design's performance by using back-annotated data. If the results do not meet design goals, return to step 3. If the results meet your design goals, you are finished with the design cycle.

Running Design Compiler

This section provides the basic information you need to run Design Compiler. It includes the following sections:

- Design Compiler Modes
- Design Compiler Interfaces
- Setup Files
- Starting Design Compiler
- Exiting Design Compiler
- Getting Command Help
- Using Command Log Files
- Using Script Files
- Working with Licenses

Design Compiler Modes

Design Compiler provides two memory management modes: XG mode and DB mode. XG mode uses optimized memory management techniques that increase the tool capacity and can reduce runtime. DB mode uses the original Design Compiler memory management techniques. In general, `dc_shell` behaves the same in DB mode and XG mode, but XG mode can provide you with reduced memory consumption and runtime.

The Design Compiler documentation set discusses features supported in DB mode. For features supported in XG mode and differences between DB mode and XG mode, see the *XG Mode User Guide*.

Design Compiler Interfaces

Design Compiler offers two command environments for synthesis and timing analysis: the `dc_shell` command-line interface and the graphical user interface (GUI). The `dc_shell` command-line interface is a text-only environment in which you enter commands at the `dc_shell` prompt. Design Vision is the graphical user interface to the Synopsys synthesis environment for visualizing design data and analysis results. Design Compiler provides two command-line modes:

- `dcsh`

This command-line mode uses a command language specific to Synopsys.

- `dctl`

This command-line mode uses a command language based on the tool command language (Tcl), and includes certain command extensions needed to implement specific Design Compiler functionality.

The command-line modes provide capabilities similar to UNIX command shells, including variables, conditional execution of commands, and control flow commands. You can execute Design Compiler commands in the following ways:

- By entering single commands interactively in the shell
- By running one or more command scripts, which are text files of commands
- Type single commands interactively in the command line of the Design Vision command window. You can do this to supplement the subset of Design Compiler commands available through the menu interface.

For more information on these interfaces, see the *Design Compiler Command-Line Interface Guide*.

Setup Files

When you invoke Design Compiler, it automatically executes commands in three setup files. These files have the same file name, `.synopsys_dc.setup`, but reside in different directories. The files contain commands that initialize parameters and variables, declare design libraries, and so forth.

Design Compiler reads the three `.synopsys_dc.setup` files from three directories in the following order:

1. The Synopsys root directory

2. Your home directory
3. The current working directory (the directory from which you invoke Design Compiler)

Table 2-1 Setup Files

File	Location	Function
System-wide .synopsys_dc.setup file	Synopsys root directory (\$SYNOPSYS/admin/ setup)	This file contains system variables defined by Synopsys and general Design Compiler setup information for all users at your site. Only the system administrator can modify this file.
User-defined .synopsys_dc.setup file	User home directory	This file contains variables that define your preferences for the Design Compiler working environment. The variables in this file override the corresponding variables in the systemwide setup file.
Design-specific .synopsys_dc.setup file	Working directory from which you started Design Compiler	This file contains project- or design-specific variables that affect the optimizations of all designs in this directory. To use the file, you must invoke Design Compiler from this directory. Variables defined in this file override the corresponding variables in the user-defined and systemwide setup files.

Example 2-1 shows a sample .synopsys_dc.setup file.

Example 2-1 *.synopsys_dc.setup File*

```
# Define the target technology library, symbol library,
# and link libraries
set target_library lsi_10k.db
set symbol_library lsi_10k.sdb
set synthetic_library dw_foundation.sldb
set link_library "*" $target_library $synthetic_library"
set search_path [concat $search_path ./src]
set designer "Your Name"

# Define aliases
alias h history
alias rc "report_constraint -all_violators"
```

Starting Design Compiler

Use the commands below to start Design Compiler:

- To start dc_shell in dcsh mode, enter

```
% dc_shell
```

The system prompt changes to

```
% dc_shell>
```

- To start dc_shell in the dtcl mode, enter

```
% dc_shell -tcl_mode
```

or

```
% dc_shell-t
```

The system prompt changes to

```
% dc_shell-t>
```


You can also include numerous options in these command lines, such as

- `-checkout` to access licensed features in addition to the default features checked out by the program
- `-wait` to set a wait time limit for checking out any additional licenses
- `-f` to execute a script file before displaying the initial `dc_shell` prompt
- `-x` to include a `dc_shell` statement that is executed at startup

For a detailed list of options, see the manpages for `dc_shell`.

At startup, `dc_shell` does the following tasks:

1. Creates a command log file.
2. Reads and executes the `.synopsys_dc.setup` files
3. Executes any script files or commands specified by the `-x` and `-f` options, respectively, on the command line
4. Displays the program header and `dc_shell` prompt in the window from which you invoked `dc_shell`. The program header lists all features for which your site is licensed.

Exiting Design Compiler

You can exit Design Compiler at any time and return to the operating system.

Note:

By default, `dc_shell` saves the session information in the `command.log` file. But if you change the name of the `command_log_file` variable (dcsh mode) or `sh_command_log_file` (dctcl mode) after you start the tool, session information might be lost.

Also, `dc_shell` does not automatically save the designs loaded in memory. If you want to save these designs before exiting, use the `write` command. For example,

```
dc_shell> write -hierarchy -output my_design.db
```

To exit `dc_shell`, do one of the following:

- Enter `quit`.
- Enter `exit`.
- Press Control-d, if you are running Design Compiler in interactive mode and the tool is busy.

When you exit `dc_shell`, text similar to the following appears (the memory and the CPU numbers reflect your actual usage):

```
Memory usage for this session 1373 Kbytes.  
CPU usage for this session 4 seconds.
```

```
Thank you ...
```

Getting Command Help

Design Compiler provides three levels of command help:

- A list of commands
- Command usage help

- Topic help

To get a list of all `dc_shell` commands, enter one of the following commands (depending on your shell mode):

```
dc_shell> list -commands
```

```
dc_shell-t> help
```

In `dctcl` mode, the `help` command without options displays the commands with their command summaries.

To get help about a particular `dc_shell` command, enter the command name with the `-help` option. The syntax is

```
dc_shell> command_name -help
```

To get topic help in `dc_shell`, enter

```
dc_shell> man topic
```

where *topic* is the name of a shell command, variable, or variable group.

Using the `man` command (or `help` command in `dcsh` mode), you can display the man pages for the topic while you are interactively running Design Compiler.

Using Command Log Files

The command log file records the `dc_shell` commands processed by Design Compiler, including setup file commands and variable assignments. By default, Design Compiler writes the command log to a file called `command.log` in the directory from which you invoked `dc_shell`.

You can change the name of the `command.log` file by using the `command_log_file` variable (dcsh mode) or `sh_command_log_file` variable (dctcl mode) in the `.synopsys_dc.setup` file. You should make any changes to these variables before you start Design Compiler. If your user-defined or project-specific `.synopsys_dc.setup` file does not contain either variable, Design Compiler automatically creates the `command.log` file.

Each Design Compiler session overwrites the command log file. To save a command log file, move it or rename it. You can use the command log file to

- Produce a script for a particular synthesis strategy
- Record the design exploration process
- Document any problems you are having

Using the Filename Log File

By default, Design Compiler writes the log of filenames that it has read to the filename log file in the directory from which you invoked `dc_shell`. You can use the filename log file to identify data files needed to reproduce an error in case Design Compiler terminates abnormally. You specify the name of the filename log file with the `filename_log_file` variable in the `.synopsys_dc.setup` file.

Using Script Files

You can create a command script file by placing a sequence of `dc_shell` commands in a text file. Any `dc_shell` command can be executed within a script file.

In dcsh mode, comments are enclosed between `/*` and `*/`. For example,

```
/* This is a comment */
```

In dctcl mode, a `#` at the beginning of a line denotes a comment. For example,

```
# This is a comment
```

To execute a script file, use one of the following commands:

- `include` (in dcsh mode)
- `source` (in dctcl mode)

When a script finishes processing, `dc_shell` returns a value of 1 if the script ran successfully or a value of 0 if the script failed.

For more information about script files, see the *Design Compiler Command-Line Interface Guide*.

Working with Licenses

In working with licenses, you need to determine what licenses are in use and know how to obtain and release licenses.

Listing the Licenses in Use

Before you check out a license, use the `license_users` command to determine which licenses are already in use. For example,

```
dc_shell> license_users  
eng1@bill Design-Compiler  
eng2@matt Design-Compiler, DC-Ultra-Opt  
2 users listed.
```

Getting Licenses

When you invoke Design Compiler, the Synopsys Common Licensing software automatically checks out the appropriate license. For example, if you read in an HDL design description, Synopsys Common Licensing checks out a license for the appropriate HDL compiler.

If you know the tools and interfaces you need, you can use the `get_license` command to check out those licenses. This ensures that each license is available when you are ready to use it. For example,

```
dc_shell> get_license HDL-Compiler
```

Once a license is checked out, it remains checked out until you release it or exit `dc_shell`.

Releasing Licenses

To release a license that is checked out to you, use the `remove_license` command. For example,

```
dc_shell> remove_license HDL-Compiler
```

Following the Basic Synthesis Flow

[Figure 2-2](#) shows the basic synthesis flow. You can use this synthesis flow in both the design exploration and design implementation stages of the high-level design flow discussed previously.

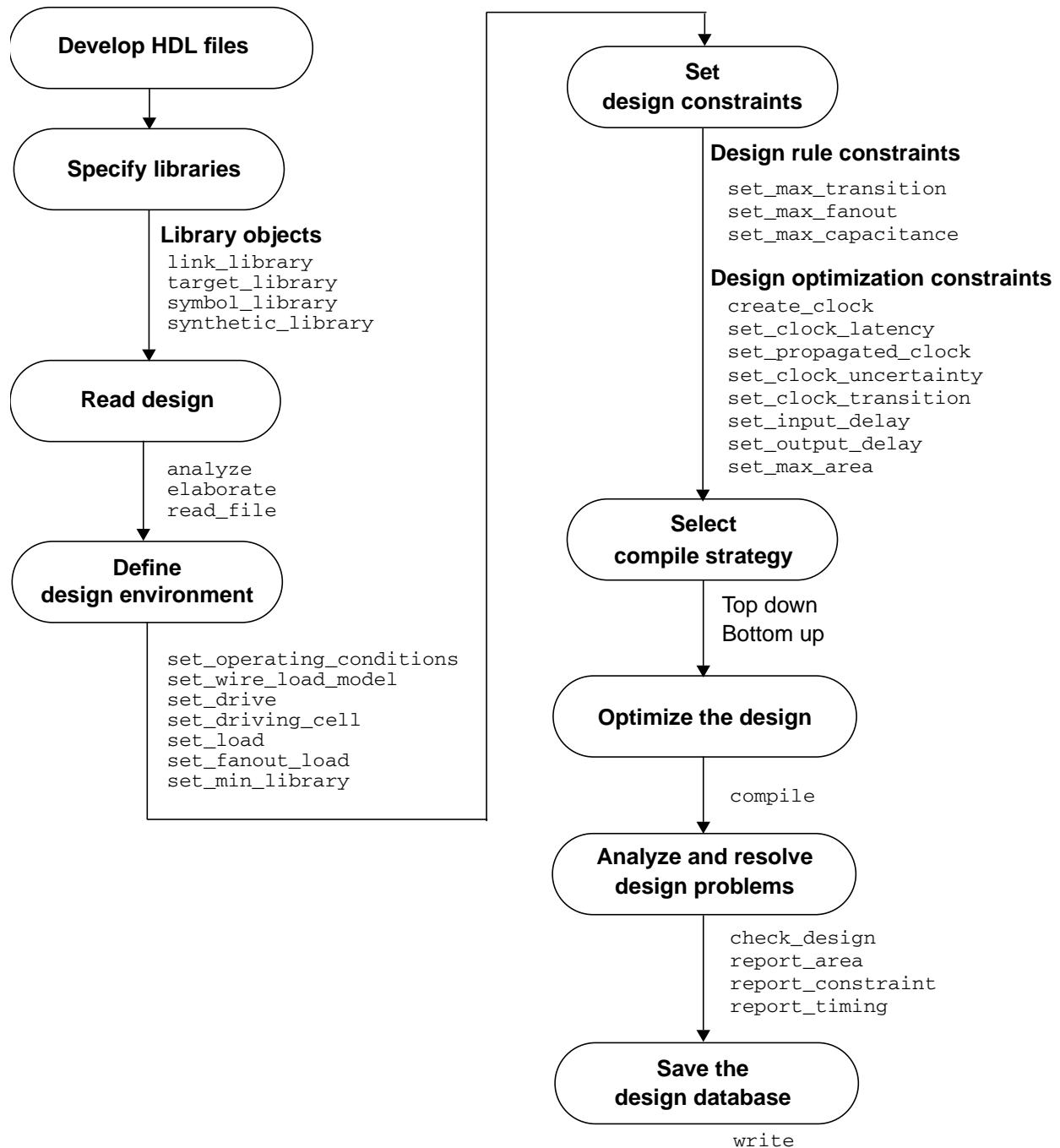
Also listed in [Figure 2-2](#) are the basic `dc_shell` commands that are commonly used in each step of the basic flow. For example, the commands `analyze`, `elaborate`, and `read_file` are used in the step that reads design files into memory. All the commands shown in [Figure 2-2](#) can take options, but no options are shown in the figure.

Note:

Under “Select Compile Strategy,” top down and bottom up are not commands. They refer to two commonly used compile strategies that use different combinations of commands.

Following [Figure 2-2](#) is a discussion of each step in the flow, including a reference to the chapter in this manual where you can find more information.

Figure 2-2 Basic Synthesis Flow



The basic synthesis flow consists of the following steps:

1. Develop HDL Files

The input design files for Design Compiler are often written using a hardware description language (HDL) such as Verilog or VHDL. These design descriptions need to be written carefully to achieve the best synthesis results possible. When writing HDL code, you need to consider design data management, design partitioning, and your HDL coding style. Partitioning and coding style directly affect the synthesis and optimization processes.

Note:

This step is included in the flow, but it is not actually a Design Compiler step. You do not create HDL files with the Design Compiler tools.

See [Chapter 3, “Preparing Design Files for Synthesis.”](#)

2. Specify Libraries

You specify the link, target, symbol, and synthetic libraries for Design Compiler by using the `link_library`, `target_library`, `symbol_library`, and `synthetic_library` commands.

The link and target libraries are technology libraries that define the semiconductor vendor’s set of cells and related information, such as cell names, cell pin names, delay arcs, pin loading, design rules, and operating conditions.

The symbol library defines the symbols for schematic viewing of the design. You need this library if you intend to use the Design Vision GUI.

In addition, you must specify any specially licensed DesignWare libraries by using the `synthetic_library` command. (You do not need to specify the standard DesignWare library.)

See [Chapter 4, “Working With Libraries.”](#)

3. Read Design

Design Compiler can read both RTL designs and gate-level netlists. Design Compiler uses HDL Compiler to read Verilog and VHDL RTL designs. It has a specialized netlist reader for reading Verilog and VHDL gate-level netlists. The specialized netlist reader reads netlists faster and uses less memory than HDL Compiler.

Design Compiler provides the following ways to read design files:

- The `analyze` and `elaborate` commands
- The `read_file` command
- The `read_vhdl` and `read_verilog` commands. These commands are derived from the `read_file -format VHDL` and `read_file -format verilog` commands.

See [Chapter 5, “Working With Designs in Memory.”](#) For detailed information on the recommended reading methods, see the HDL Compiler documentation.

4. Define Design Environment

Design Compiler requires that you model the environment of the design to be synthesized. This model comprises the external operating conditions (manufacturing process, temperature, and voltage), loads, drives, fanouts, and wire load models. It directly

influences design synthesis and optimization results. You define the design environment by using the set commands listed under this step of [Figure 2-2](#).

See [Chapter 6, “Defining the Design Environment.”](#)

5. Set Design Constraints

Design Compiler uses design rules and optimization constraints to control the synthesis of the design. Design rules are provided in the vendor technology library to ensure that the product meets specifications and works as intended. Typical design rules constrain transition times (`set_max_transition`), fanout loads (`set_max_fanout`), and capacitances (`set_max_capacitance`). These rules specify technology requirements that you cannot violate. (You can, however, specify stricter constraints.)

Optimization constraints define the design goals for timing (clocks, clock skews, input delays, and output delays) and area (maximum area). In the optimization process, Design Compiler attempts to meet these goals, but no design rules are violated by the process. You define these constraints by using commands such as those listed under this step in [Figure 2-2](#). To optimize a design correctly, you must set realistic constraints.

Note:

Design constraint settings are influenced by the compile strategy you choose. Flow steps 5 and 6 are interdependent. Compile strategies are discussed in step 6.

See [Chapter 7, “Defining Design Constraints.”](#)

6. Select Compile Strategy

The two basic compile strategies that you can use to optimize hierarchical designs are referred to as top down and bottom up.

In the top-down strategy, the top-level design and all its subdesigns are compiled together. All environment and constraint settings are defined with respect to the top-level design. Although this strategy automatically takes care of interblock dependencies, the method is not practical for large designs because all designs must reside in memory at the same time.

In the bottom-up strategy, individual subdesigns are constrained and compiled separately. After successful compilation, the designs are assigned the `dont_touch` attribute to prevent further changes to them during subsequent compile phases. Then the compiled subdesigns are assembled to compose the designs of the next higher level of the hierarchy (any higher-level design can also incorporate unmapped logic), and these designs are compiled. This compilation process is continued up through the hierarchy until the top-level design is synthesized. This method lets you compile large designs because Design Compiler does not need to load all the uncompiled subdesigns into memory at the same time. At each stage, however, you must estimate the interblock constraints, and typically you must iterate the compilations, improving these estimates, until all subdesign interfaces are stable.

Each strategy has its advantages and disadvantages, depending on your particular designs and design goals. You can use either strategy to process the entire design, or you can mix strategies, using the most appropriate strategy for each subdesign.

Note:

The compile strategy you choose affects your choice of design constraints and the values you set. Flow steps 5 and 6 are interdependent. Design constraints are discussed in step 5.

See [Chapter 8, “Optimizing the Design.”](#)

7. Optimize the Design

You use the `compile` command to invoke the Design Compiler synthesis and optimization processes. Several compile options are available. In particular, the `map_effort` option can be set to low, medium, or high.

In a preliminary compile, when you want to get a quick idea of design area and performance, you set `map_effort` to low. In a default compile, when you are performing design exploration, you use the medium `map_effort` option. Because this option is the default, you do not need to specify `map_effort` in the `compile` command. In a final design implementation compile, you might want to set `map_effort` to high. You should use this option judiciously, however, because the resulting compile process is CPU intensive. Often setting `map_effort` to medium is sufficient.

See [Chapter 8, “Optimizing the Design.”](#)

8. Analyze and Resolve Design Problems

Design Compiler can generate numerous reports on the results of a design synthesis and optimization, for example, area, constraint, and timing reports. You use reports to analyze and resolve any design problems or to improve synthesis results. You can use the `check_design` command to check the synthesized design for consistency. Other `check_` commands are available.

See [Chapter 9, “Analyzing and Resolving Design Problems.”](#)

9. Save the Design Database

You use the `write` command to save the synthesized designs. Remember that Design Compiler does not automatically save designs before exiting.

You can also save in a script file the design attributes and constraints used during synthesis. Script files are ideal for managing your design attributes and constraints.

See the section [“Exiting Design Compiler” on page 2-11](#) and see the chapters on using script files in the *Design Compiler Command-Line Interface Guide*.

A Design Compiler Session Example

[Example 2-2 on page 2-25](#) shows a simple dcsch script that performs a top-down compile run ([Example 2-3 on page 2-26](#) shows the same script in dctcl syntax). It uses the basic synthesis flow. The script contains comments that identify each of the steps in the flow. Some of the script command options and arguments have not yet been explained in this manual. Nevertheless, from the previous discussion of the basic synthesis flow, you can begin to understand this example of a top-down compile. The remaining chapters will help you understand these commands in detail.

Note:

Only the `set_driving_cell` command is not discussed in the section on basic synthesis design flow. The `set_driving_cell` command is an alternative way to set the external drives on the ports of the design to be synthesized.

Example 2-2 Top-Down Compile Script in dcsh Mode

```
/* specify the libraries */
target_library = my_lib.db
symbol_library = my_lib.sdb
link_library = "*" + target_library

/* read the design */
read_file -format verilog Adder16.v

/* define the design environment */
set_operating_conditions WCCOM
set_wire_load_model "10x10"
set_load 2.2 sout
set_load 1.5 cout
set_driving_cell -lib_cell FD1 all_inputs()
set_drive 0 clk

/* set the optimization constraints */
create_clock clk -period 10
set_input_delay -max 1.35 -clock clk {ain, bin}
set_input_delay -max 3.5 -clock clk cin
set_output_delay -max 2.4 -clock clk cout
set_max_area 0

/* map and optimize the design */
compile

/* analyze and debug the design */
report_constraint -all_violators
report_area

/* save the design database */
write -format db -hierarchy -output Adder16.db
```

Example 2-3 Top-Down Compile Script in dctcl Mode

```
/* specify the libraries */
set target_library my_lib.db
set symbol_library my_lib.sdb
set link_library [list "*" $target_library]

/* read the design */
read_verilog Adder16.v

/* define the design environment */
set_operating_conditions WCCOM
set_wire_load_model "10x10"
set_load 2.2 sout
set_load 1.5 cout
set_driving_cell -lib_cell FD1 [all_inputs]
set_drive 0 clk

/* set the optimization constraints */
create_clock clk -period 10
set_input_delay -max 1.35 -clock clk {ain bin}
set_input_delay -max 3.5 -clock clk cin
set_output_delay -max 2.4 -clock clk cout
set_max_area 0

/* map and optimize the design */
compile

/* analyze and debug the design */
report_constraint -all_violators
report_area

/* save the design database */
write -format db -hierarchy -output Adder16.db
```

You can execute these commands in any of the following ways:

- Enter `dc_shell` and type each command in the order shown in the example.
- Enter `dc_shell` and execute the script file, using the `include` command (`dcsh` mode) or the `source` command (`dctcl` mode).

For example, if you are running Design Compiler and the script is in a file called run.scr, you can execute the script file by entering one of the following commands (depending on your shell mode):

```
dc_shell> include run.scr
```

```
dc_shell-t> source run.scr
```

- Run the script from the UNIX command line by using the `-f` option of the `dc_shell` command.

For example, if the script is in a file called run.scr, you can invoke Design Compiler and execute the script file from the UNIX prompt by entering one of the following commands (depending on your shell mode):

```
% dc_shell -f run.scr
```

```
% dc_shell-t -f run.scr
```


3

Preparing Design Files for Synthesis

Designs (that is, design descriptions) are stored in design files, which can be ASCII or .db format. Design files must have unique names. If a design is hierarchical, each subdesign refers to another design file, which must also have a unique name. Note, however, that different design files can contain subdesigns with identical names.

This chapter contains the following sections:

- [Managing the Design Data](#)
- [Partitioning for Synthesis](#)
- [HDL Coding for Synthesis](#)

Managing the Design Data

Use systematic organizational methods to manage the design data. Two basic elements of managing design data are design data control and data organization.

Controlling the Design Data

As new versions of your design are created, you must maintain some archival and record keeping method that provides a history of the design evolution and that lets you restart the design process if data is lost. Establishing controls for data creation, maintenance, overwriting, and deletion is a fundamental design management issue.

Establishing file-naming conventions is one of the most important rules for data creation. [Table 3-1](#) lists the recommended file name extensions for each design data type.

Table 3-1 File Name Extensions

Design data type	Extension	Description
Design source code	.v	Verilog
	.vhd	VHDL
	.edif	EDIF
Synthesis scripts	.con	Constraints
	.scr	Script
Reports and logs	.rpt	Report
	.log	Log
Design database	.db	Synopsys database format

Organizing the Design Data

Establishing and adhering to a method of organizing data are more important than the method you choose. After you place the essential design data under a consistent set of controls, you can create a meaningful data organization. To simplify data exchanges and data searches, designers should adhere to this data organization system.

You can use a hierarchical directory structure to address data organization issues. Your compile strategy will influence your directory structure. The following figures show directory structures based on the top-down compile strategy ([Figure 3-1](#)) and the bottom-up compile strategy ([Figure 3-2](#)). For details about compile strategies, see [“Selecting and Using a Compile Strategy” on page 8-7](#).

Figure 3-1 Top-Down Compile Directory Structure

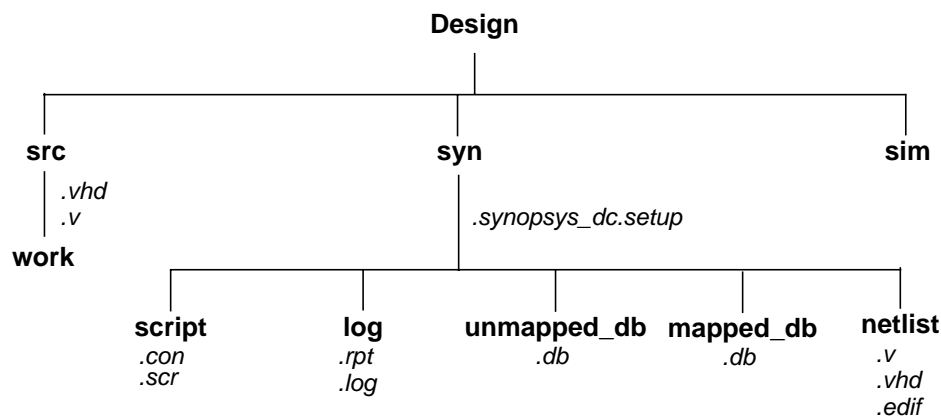
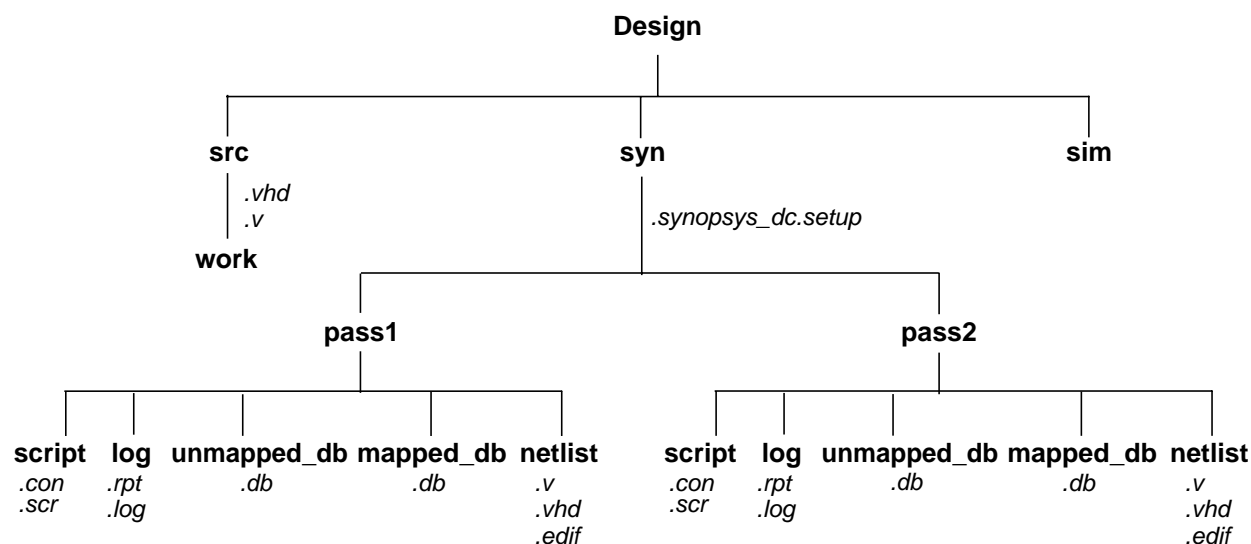


Figure 3-2 Bottom-Up Compile Directory Structure



Partitioning for Synthesis

Partitioning a design effectively can enhance the synthesis results, reduce compile time, and simplify the constraint and script files.

Partitioning affects block size, and although Design Compiler has no inherent block size limit, you should be careful to control block size. If you make blocks too small, you can create artificial boundaries that restrict effective optimization. If you create very large blocks, compile runtimes can be lengthy.

Use the following strategies to partition your design and improve optimization and runtimes:

- Partition for design reuse.
- Keep related combinational logic together.
- Register the block outputs.

- Partition by design goal.
- Partition by compile technique.
- Keep sharable resources together.
- Keep user-defined resources with the logic they drive.
- Isolate special functions, such as pads, clocks, boundary scans, and asynchronous logic.

The following sections describe each of these strategies.

Partitioning for Design Reuse

Design reuse decreases time to market by reducing the design, integration, and testing effort.

When reusing existing designs, partition the design to enable instantiation of the designs.

To enable designs to be reused, follow these guidelines during partitioning and block design:

- Thoroughly define and document the design interface.
- Standardize interfaces whenever possible.
- Parameterize the HDL code.

Keeping Related Combinational Logic Together

By default, Design Compiler cannot move logic across hierarchical boundaries. Dividing related combinational logic into separate blocks introduces artificial barriers that restrict logic optimization.

For best results, apply these strategies:

- Group related combinational logic and its destination register together.

When working with the complete combinational path, Design Compiler has the flexibility to merge logic, resulting in a smaller, faster design. Grouping combinational logic with its destination register also simplifies the timing constraints and enables sequential optimization.

- Eliminate glue logic.

Glue logic is the combinational logic that connects blocks. Moving this logic into one of the blocks improves synthesis results by providing Design Compiler with additional flexibility. Eliminating glue logic also reduces compile time, because Design Compiler has fewer logic levels to optimize.

For example, assume that you have a design containing three combinational clouds on or near the critical path. [Figure 3-3](#) shows poor partitioning of this design. Each of the combinational clouds occurs in a separate block, so Design Compiler cannot fully exploit its combinational optimization techniques.

Figure 3-3 Poor Partitioning of Related Logic

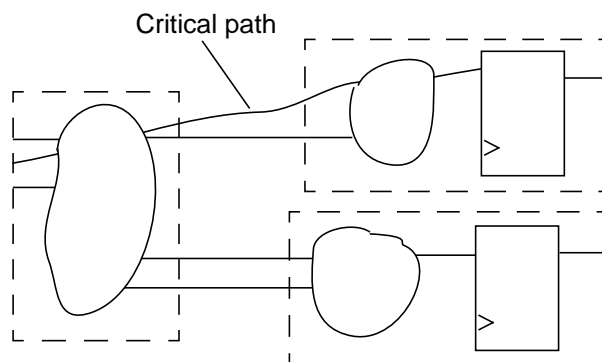
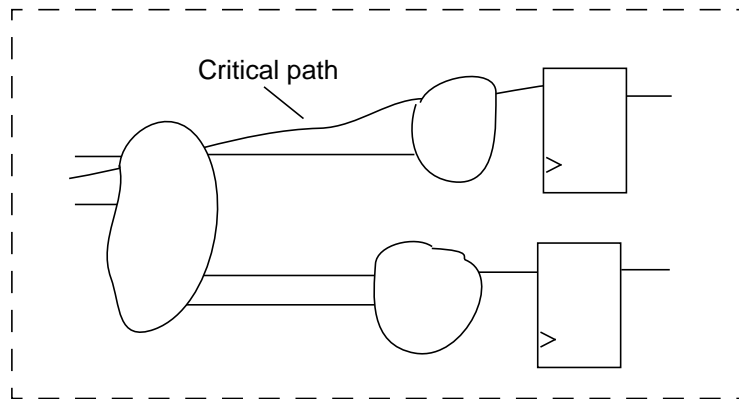


Figure 3-4 shows the same design with no artificial boundaries. In this design, Design Compiler has the flexibility to combine related functions in the combinational clouds.

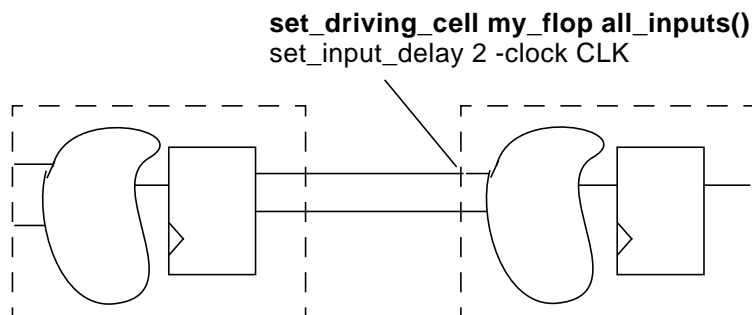
Figure 3-4 Keeping Related Logic in the Same Block



Registering Block Outputs

To simplify the constraint definitions, make sure that registers drive the block outputs, as shown in Figure 3-5.

Figure 3-5 Registering All Outputs



This method enables you to constrain each block easily because

- The drive strength on the inputs to an individual block always equals the drive strength of the average input drive

- The input delays from the previous block always equal the path delay through the flip-flop

Because no combinational-only paths exist when all outputs are registered, time budgeting the design and using the `set_output_delay` command are easier. Given that one clock cycle occurs within each module, the constraints are simple and identical for each module.

This partitioning method can improve simulation performance. With all outputs registered, a module can be described with only edge-triggered processes. The sensitivity list contains only the clock and, perhaps, a reset pin. A limited sensitivity list speeds simulation by having the process triggered only once in each clock cycle.

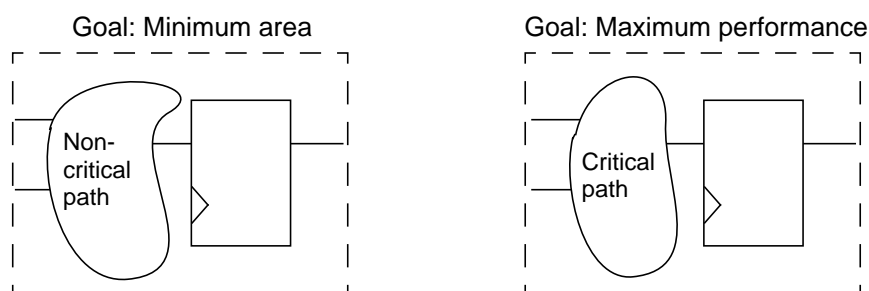
Partitioning by Design Goal

Partition logic with different design goals into separate blocks. Use this method when certain parts of a design are more area and timing critical than other parts.

To achieve the best synthesis results, isolate the noncritical speed constraint logic from the critical speed constraint logic. By isolating the noncritical logic, you can apply different constraints, such as a maximum area constraint, on the block.

[Figure 3-6](#) shows how to separate logic with different design goals.

Figure 3-6 Blocks With Different Constraints



Partitioning by Compile Technique

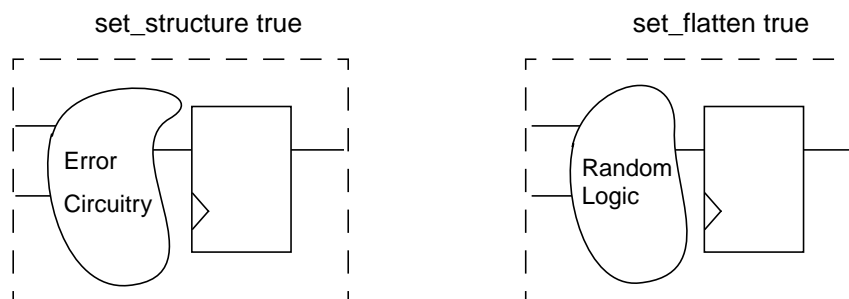
Partition logic that requires different compile techniques into separate blocks. Use this method when the design contains highly structured logic along with random logic.

- Highly structured logic, such as error detection circuitry, which usually contains large exclusive OR trees, is better suited to structuring.
- Random logic is better suited to flattening.

For more information on these two compile techniques, see [“Logic-Level Optimization” on page 8-3](#).

[Figure 3-7](#) shows the logic separated into different blocks.

Figure 3-7 Blocks With Different Compile Techniques



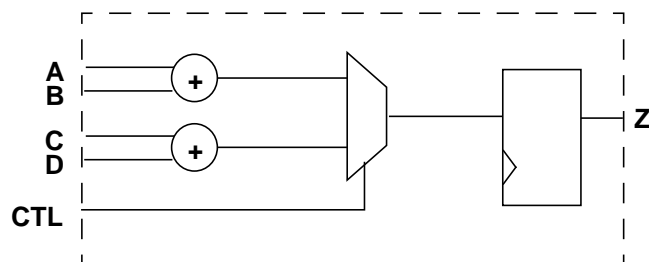
Keeping Sharable Resources Together

Design Compiler can share large resources, such as adders or multipliers, but resource sharing can occur only if the resources belong to the same VHDL process or Verilog always block.

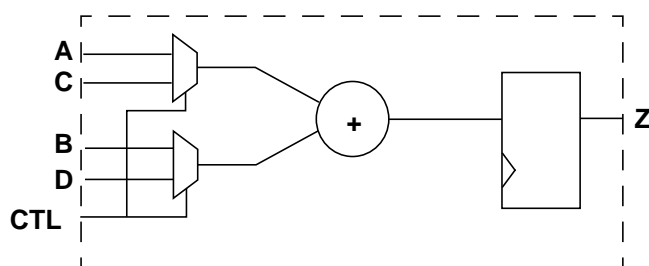
For example, if two separate adders have the same destination path and have multiplexed outputs to that path, keep the adders in one VHDL process or Verilog always block. This approach allows Design Compiler to share resources (using one adder instead of two) if the constraints allow sharing. [Figure 3-8](#) shows possible implementations of a logic example.

Figure 3-8 Keeping Sharable Resources in the Same Process

Unshared Resources



Shared Resources



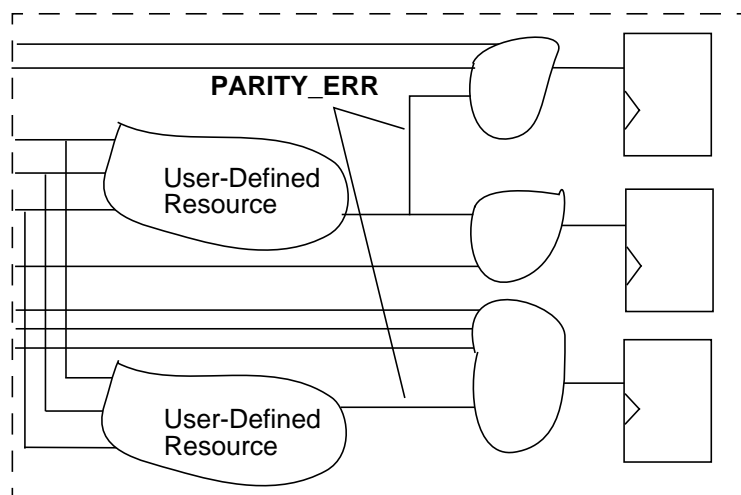
For more information about resource sharing, see the HDL Compiler documentation.

Keeping User-Defined Resources With the Logic They Drive

User-defined resources are user-defined functions, procedures, or macro cells, or user-created DesignWare components. Design Compiler cannot automatically share or create multiple instances of user-defined resources. Keeping these resources with the logic they drive, however, gives you the flexibility to split the load by manually inserting multiple instantiations of a user-defined resource if timing goals cannot be achieved with a single instantiation.

Figure 3-9 illustrates splitting the load by multiple instantiation when the load on the signal PARITY_ERR is too heavy to meet constraints.

Figure 3-9 Duplicating User-Defined Resources

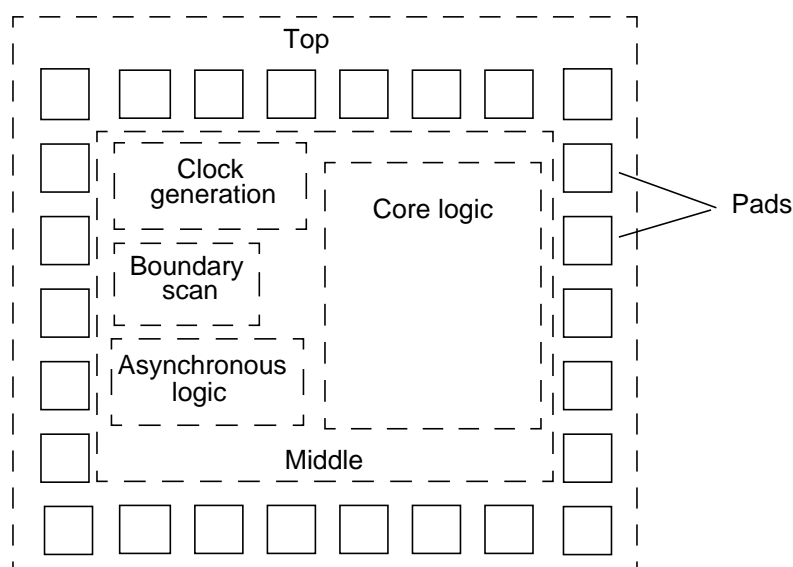


Isolating Special Functions

Isolate special functions (such as I/O pads, clock generation circuitry, boundary-scan logic, and asynchronous logic) from the core logic.

[Figure 3-10](#) shows the recommended partitioning for the top level of the design.

Figure 3-10 Recommended Top-Level Partitioning



The top level of the design contains the I/O pad ring and a middle level of hierarchy that contains submodules for the boundary-scan logic, the clock generation circuitry, the asynchronous logic, and the core logic. The middle level of hierarchy exists to allow the flexibility to instantiate I/O pads. Isolation of the clock generation circuitry enables instantiation and careful simulation of this module. Isolation of the asynchronous logic helps confine testability problems and static timing analysis problems to a small area.

HDL Coding for Synthesis

HDL coding is the foundation for synthesis because it implies the initial structure of the design. When writing your HDL source code, always consider the hardware implications of the code. A good coding style can generate smaller and faster designs. This section provides information to help you write efficient code so that you can achieve your design target in the shortest possible time.

Topics include

- Writing technology-independent HDL
- Using HDL constructs
- Writing effective code

Writing Technology-Independent HDL

The goal of high-level design that uses a completely automatic synthesis process is to have no instantiated gates or flip-flops. If you meet this goal, you will have readable, concise, and portable high-level HDL code that can be transferred to other vendors or to future processes.

In some cases, the HDL Compiler tool requires compiler directives to provide implementation information while still maintaining technology independence. In Verilog, compiler directives begin with the characters `//` or `/*`. In VHDL, compiler directives begin with the two hyphens (`--`) followed by *pragma* or *synopsys*. For more information, see the HDL Compiler documentation.

The following sections discuss various methods for keeping your HDL code technology independent.

Inferring Components

HDL Compiler provides the capability to infer the following components:

- Multiplexers
- Registers
- Three-state drivers

- Multibit components

These inference capabilities are discussed in the following pages. For additional information and examples, see the HDL Compiler documentation.

Inferring Multiplexers. HDL Compiler can infer a generic multiplexer cell (MUX_OP) from case statements in your HDL code.. If your target technology library contains at least a 2-to-1 multiplexer cell, Design Compiler maps the inferred MUX_OPs to multiplexer cells in the target technology library. Design Compiler determines the MUX_OP implementation during compile based on the design constraints. For information about how Design Compiler maps MUX_OPs to multiplexers, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*.

Use the `infer_mux` compiler directive to control multiplexer inference. When attached to a block, the `infer_mux` directive forces multiplexer inference for all case statements in the block. When attached to a case statement, the `infer_mux` directive forces multiplexer inference for that specific case statement.

Inferring Registers. Register inference allows you to specify technology-independent sequential logic in your designs. A register is a simple, 1-bit memory device, either a latch or a flip-flop. A latch is a level-sensitive memory device. A flip-flop is an edge-triggered memory device.

HDL Compiler infers a D latch whenever you do not specify the resulting value for an output under all conditions, as in an incompletely specified if or case statement. HDL Compiler can also infer SR latches and master-slave latches.

HDL Compiler infers a D flip-flop whenever the sensitivity list of a Verilog always block or VHDL process includes an edge expression (a test for the rising or falling edge of a signal). HDL Compiler can also infer JK flip-flops and toggle flip-flops.

Mixing Register Types. For best results, restrict each Verilog always block or VHDL process to a single type of register inferencing: latch, latch with asynchronous set or reset, flip-flop, flip-flop with asynchronous set or reset, or flip-flop with synchronous set or reset.

Be careful when mixing rising- and falling-edge-triggered flip-flops in your design. If a module infers both rising- and falling-edge-triggered flip-flops and the target technology library does not contain a falling-edge-triggered flip-flop, Design Compiler generates an inverter in the clock tree for the falling-edge clock.

Inferring Registers Without Control Signals. For inferring registers without control signals, make the data and clock pins controllable from the input ports or through combinational logic. If a gate-level simulator cannot control the data or clock pins from the input ports or through combinational logic, the simulator cannot initialize the circuit, and the simulation fails.

Inferring Registers With Control Signals. You can initialize or control the state of a flip-flop by using either an asynchronous or a synchronous control signal.

For inferring asynchronous control signals on latches, use the `async_set_reset` compiler directive (attribute in VHDL) to identify the asynchronous control signals. HDL Compiler automatically identifies asynchronous control signals when inferring flip-flops.

For inferring synchronous resets, use the `sync_set_reset` compiler directive (attribute in VHDL) to identify the synchronous controls.

Inferring Three-State Drivers. Assign the high-impedance value (1'bz in Verilog, 'Z' in VHDL) to the output pin to have Design Compiler infer three-state gates. Three-state logic reduces the testability of the design and makes debugging difficult. Where possible, replace three-state buffers with a multiplexer.

Never use high-impedance values in a conditional expression. HDL Compiler always evaluates expressions compared to high-impedance values as false, which can cause the gate-level implementation to behave differently from the RTL description.

For additional information about three-state inference, see the HDL Compiler documentation.

Inferring Multibit Components. Multibit inference allows you to map multiplexers, registers, and three-state drivers to regularly structured logic or multibit library cells. Using multibit components can have the following results:

- Smaller area and delay, due to shared transistors and optimized transistor-level layout
- Reduced clock skew in sequential gates
- Lower power consumption by the clock in sequential banked components
- Improved regular layout of the data path

Multibit components might not be efficient in the following instances:

- As state machine registers
- In small bused logic that would benefit from single-bit design

You must weigh the benefits of multibit components against the loss of optimization flexibility when deciding whether to map to multibit or single-bit components.

Attach the `infer_multibit` compiler directive to bused signals to infer multibit components. You can also change between a single-bit and a multibit implementation after optimization by using the `create_multibit` and `remove_multibit` commands.

For more information about how Design Compiler handles multibit components, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*.

Using Synthetic Libraries

To help you achieve optimal performance, Synopsys supplies a synthetic library. This library contains efficient implementations for adders, incrementers, comparators, and signed multipliers.

Design Compiler selects a synthetic component to meet the given constraints. After Design Compiler assigns the synthetic structure, you can always change to another type of structure by modifying your constraints. If you ungroup the synthetic cells or write the netlist to a text file, however, Design Compiler can no longer recognize the synthetic component and cannot perform implementation reselection.

The HDL Compiler documentation contains additional information about using compiler directives to control synthetic component use. The *DesignWare Foundation Library Databook* volumes contain additional information about synthetic libraries and provide examples of how to infer and instantiate synthetic components.

[Example 3-1](#) and [Example 3-2](#) use the label, ops, map_to_module, and implementation compiler directives to implement a 32-bit carry-lookahead adder.

Example 3-1 32-Bit Carry-Lookahead Adder (Verilog)

```
module add32 (a, b, cin, sum, cout);
    input [31:0] a, b;
    input cin;
    output [31:0] sum;
    output cout;
    reg [33:0] temp;

    always @(a or b or cin)
        begin : add1
            /* synopsys resource r0:
               ops = "A1",
               map_to_module = "DW01_add",
               implementation = "cla"; */
            temp = ({1'b0, a, cin} + // synopsys label A1
                   {1'b0, b, 1'b1});
        end

    assign {cout, sum} = temp[33:1];

endmodule
```

Example 3-2 32-Bit Carry-Lookahead Adder (VHDL)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

library synopsys;
use synopsys.attributes.all;

entity add32 is
    port (a,b : in std_logic_vector (31 downto 0);
          cin : in std_logic;
          sum : out std_logic_vector (31 downto 0);
          cout: out std_logic);
end add32;

architecture rtl of add32 is
    signal temp_signed : SIGNED (33 downto 0);
    signal op1, op2, temp : STD_LOGIC_VECTOR (33 downto 0);
    constant COUNT : UNSIGNED := "01";

begin
    infer: process ( a, b, cin )
        constant r0 : resource := 0;
        attribute ops of r0 : constant is "A1";
        attribute map_to_module of r0 : constant is "DW01_add";
        attribute implementation of r0 : constant is "cla";

        begin
            op1 <= '0' & a & cin;
            op2 <= '0' & b & '1';
            temp_signed <= SIGNED(op1) + SIGNED(op2); -- pragma
label A1
            temp <= STD_LOGIC_VECTOR(temp_signed);

            cout <= temp(33);
            sum <= temp(32 downto 1);
        end process infer;
    end rtl;
```

Designing State Machines

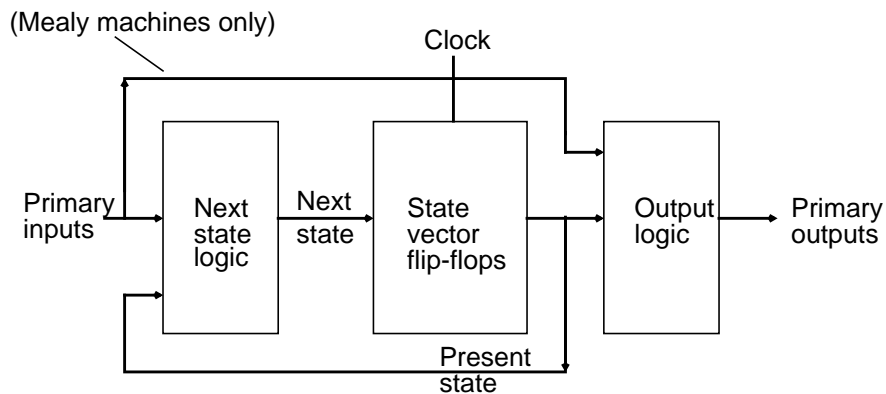
You can specify a state machine by using several different formats:

- Verilog
- VHDL
- State table
- PLA

If you use the `state_vector` and `enum` compiler directives in your HDL code, Design Compiler can extract the state table from a netlist. In the state table format, Design Compiler does not retain the `case`, `casez`, and `parallel_case` information. Design Compiler does not optimize invalid input combinations and mutually exclusive inputs.

Figure 3-11 shows the architecture for a finite state machine.

Figure 3-11 Finite State Machine Architecture



Using an extracted state table provides the following benefits:

- State minimization can be performed.
- Tradeoffs between different encoding styles can be made.

- Don't care conditions can be used without flattening the design.
- Don't care state codes are automatically derived.

For information about extracting state machines and changing encoding styles, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*.

Using HDL Constructs

The following sections provide information and guidelines about the following specific HDL constructs:

- General HDL constructs
- Verilog macro definitions
- VHDL port definitions

General HDL Constructs

The information in this section applies to both Verilog and VHDL.

Sensitivity Lists. You should completely specify the sensitivity list for each Verilog always block or VHDL process. Incomplete sensitivity lists (shown in the following examples) can result in simulation mismatches between the HDL and the gate-level design.

Example 3-3 Incomplete Sensitivity List (Verilog)

```
always @ (A)
    C <= A | B;
```


Example 3-4 Incomplete Sensitivity List (VHDL)

```
process (A)
  C <= A or B;
```

Value Assignments. Both Verilog and VHDL support the use of immediate and delayed value assignments in the RTL code. The hardware generated by immediate value assignments—implemented by Verilog blocking assignments (=) and VHDL variables (:=)—is dependent on the ordering of the assignments. The hardware generated by delayed value assignments—implemented by Verilog nonblocking assignments (<=) and VHDL signals (<=)—is independent of the ordering of the assignments.

For the most intuitive results,

- Use immediate value assignments within sequential Verilog always blocks or VHDL processes
- Use delayed value assignments within combinational Verilog always blocks or VHDL processes

if Statements. When an if statement used in a Verilog always block or VHDL process as part of a continuous assignment does not include an else clause, Design Compiler creates a latch. The following examples show if statements that generate latches during synthesis.

Example 3-5 Incorrect if Statement (Verilog)

```
if ((a == 1) && (b == 1))
  z = 1;
```

Example 3-6 Incorrect if Statement (VHDL)

```
if (a = '1' and b = '1') then
    z <= '1';
end if;
```

case Statements. If your if statement contains more than three conditions, consider using the case statement to improve the parallelism of your design and the clarity of your code. The following examples use the case statement to implement a 3-bit decoder.

Example 3-7 Using the case Statement (Verilog)

```
case ({a, b, c})
    3'b000: z = 8'b00000001;
    3'b001: z = 8'b00000010;
    3'b010: z = 8'b00000100;
    3'b011: z = 8'b00001000;
    3'b100: z = 8'b00010000;
    3'b101: z = 8'b00100000;
    3'b110: z = 8'b01000000;
    3'b111: z = 8'b10000000;
    default: z = 8'b00000000;
endcase
```

Example 3-8 Using the case Statement (VHDL)

```
case_value := a & b & c;
CASE case_value IS
    WHEN "000" =>
        z <= "00000001";
    WHEN "001" =>
        z <= "00000010";
    WHEN "010" =>
        z <= "00000100";
    WHEN "011" =>
        z <= "00001000";
    WHEN "100" =>
        z <= "00010000";
    WHEN "101" =>
        z <= "00100000";
    WHEN "110" =>
        z <= "01000000";
    WHEN "111" =>
        z <= "10000000";
    WHEN OTHERS =>
        z <= "00000000";
END CASE;
```

An incomplete case statement results in the creation of a latch. VHDL does not support incomplete case statements. In Verilog you can avoid latch inference by using either the default clause or the `full_case` compiler directive.

Although both the `full_case` directive and the default clause prevent latch inference, they have different meanings. The `full_case` directive asserts that all valid input values have been specified and no default clause is necessary. The default clause specifies the output for any undefined input values.

For best results, use the default clause instead of the `full_case` directive. If the unspecified input values are don't care conditions, using the default clause with an output value of x can generate a smaller implementation.

If you use the `full_case` directive, the gate-level simulation might not match the RTL simulation whenever the case expression evaluates to an unspecified input value. If you use the default clause, simulation mismatches can occur only if you specified don't care conditions and the case expression evaluates to an unspecified input value.

Constant Definitions. Use the Verilog ``define` statement or the VHDL constant statement to define global constants. Keep global constant definitions in a separate file. Use parameters (Verilog) or generics (VHDL) to define local constants.

[Example 3-9](#) shows a Verilog code fragment that includes a global ``define` statement and a local parameter. [Example 3-10](#) shows a VHDL code fragment that includes a global constant and a local generic.

Example 3-9 Using Macros and Parameters (Verilog)

```
// Define global constant in def_macro.v
`define WIDTH 128

// Use global constant in reg128.v
reg regfile[WIDTH-1:0];

// Define and use local constant in module foo
module foo (a, b, c);
    parameter WIDTH=128;
    input [WIDTH-1:0] a, b;
    output [WIDTH-1:0] c;
```

Example 3-10 Using Global Constants and Generics (VHDL)

```
-- Define global constant in synthesis_def.vhd
constant WIDTH : INTEGER := 128;

-- Include global constants
library my_lib;
USE my_lib.synthesis_def.all;

-- Use global constant in entity foo
entity fool is
    port (a,b : in std_logic_vector(WIDTH-1 downto 0);
          c: out std_logic_vector(WIDTH-1 downto 0));
end foo;

-- Define and use local constant in entity foo
entity foo is
    generic (WIDTH_VAR : INTEGER := 128);
    port (a,b : in std_logic_vector(WIDTH-1 downto 0);
          c: out std_logic_vector(WIDTH-1 downto 0));
end foo;
```

Using Verilog Macro Definitions

In Verilog, macros are implemented using the ``define` statement. Follow these guidelines for ``define` statements:

- Use ``define` statements only to declare constants.
- Keep ``define` statements in a separate file.
- Do not use nested ``define` statements.

Reading a macro that is nested more than twice is difficult. To make your code readable, do not use nested ``define` statements.

- Do not use ``define` inside module definitions.

When you use a ``define` statement inside a module definition, the local macro and the global macro have the same reference name but different values. Use parameters to define local constants.

Using VHDL Port Definitions

When defining ports in VHDL source code, observe these guidelines:

- Use the `STD_LOGIC` and `STD_LOGIC_VECTOR` packages.

By using `STD_LOGIC`, you avoid the need for type conversion functions on the synthesized design.

- Do not use the buffer port mode.

When you declare a port as a buffer, the port must be used as a buffer throughout the hierarchy. To simplify synthesis, declare the port as an output, then define an internal signal that drives the output port.

Writing Effective Code

This section provides guidelines for writing efficient, readable HDL source code for synthesis. The guidelines cover

- Identifiers
- Expressions
- Functions
- Modules

Guidelines for Identifiers

A good identifier name conveys the meaning of the signal, the value of a variable, or the function of a module; without this information, the hardware descriptions are difficult to read.

Observe the following naming guidelines to improve the readability of your HDL source code:

- Ensure that the signal name conveys the meaning of the signal or the value of a variable without being verbose.

For example, assume that you have a variable that represents the floating point opcode for rs1. A short name, such as `frs1`, does not convey the meaning to the reader. A long name, such as `floating_pt_opcode_rs1`, conveys the meaning, but its length might make the source code difficult to read. Use a name such as `fpop_rs1`, which meets both goals.

- Use a consistent naming style for capitalization and to distinguish separate words in the name.

Commonly used styles include C, Pascal, and Modula.

- C style uses lowercase names and separates words with an underscore, for example, `packet_addr`, `data_in`, and `first_grant_enable`.
- Pascal style capitalizes the first letter of the name and first letter of each word, for example, `PacketAddr`, `DataIn`, and `FirstGrantEnable`.
- Modula style uses a lowercase letter for the first letter of the name and capitalizes the first letter of subsequent words, for example, `packetAddr`, `dataIn`, and `firstGrantEnable`.

Choose one convention and apply it consistently.

- Avoid confusing characters.

Some characters (letters and numbers) look similar and are easily confused, for example, O and 0 (zero); l and 1 (one).

- Avoid reserved words.

- Use the noun or noun followed by verb form for names, for example, AddrDecode, DataGrant, PCI_interrupt.
- Add a suffix to clarify the meaning of the name.

Table 3-2 shows common suffixes and their meanings.

Table 3-2 Signal Name Suffixes and Their Meanings

Suffix	Meaning
_clk	Clock signal
_next	Signal before being registered
_n	Active low signal
_z	Signal that connects to a three-state output
_f	Register that uses an active falling edge
_xi	Primary chip input
_xo	Primary chip output
_xod	Primary chip open drain output
_xz	Primary chip three-state output
_xbio	Primary chip bidirectional I/O

Guidelines for Expressions

Observe the following guidelines for expressions:

- Use parentheses to indicate precedence.

Expression operator precedence rules are confusing, so you should use parentheses to make your expression easy to read. Unless you are using DesignWare resources, parentheses have little effect on the generated logic. An example of a logic expression without parentheses that is difficult to read is


```
bus_select = a ^ b & c~^d|b^~e&^f[1:0];
```

- Replace repetitive expressions with function calls or continuous assignments.

If you use a particular expression more than two or three times, consider replacing the expression with a function or a continuous assignment that implements the expression.

Guidelines for Functions

Observe these guidelines for functions:

- Do not use global references within a function.

In procedural code, a function is evaluated when it is called. In a continuous assignment, a function is evaluated when any of its declared inputs changes.

Avoid using references to nonlocal names within a function because the function might not be reevaluated if the nonlocal value changes. This can cause a simulation mismatch between the HDL description and the gate-level netlist. For example, the following Verilog function references the nonlocal name `byte_sel`:

```
function byte_compare;
    input [15:0] vector1, vector2;
    input [7:0] length;

    begin
        if (byte_sel)
            // compare the upper byte
        else
            // compare the lower byte
        ...
    end
endfunction // byte_compare
```

- Be aware that the local storage for tasks and functions is static.

Formal parameters, outputs, and local variables retain their values after a function has returned. The local storage is reused each time the function is called. This storage can be useful for debugging, but storage reuse also means that functions and tasks cannot be called recursively.

- Be careful when using component implication.

You can map a function to a specific implementation by using the `map_to_module` and `return_port_name` compiler directives. Simulation uses the contents of the function. Synthesis uses the gate-level module in place of the function. When you are using component implication, the RTL model and the gate-level model might be different. Therefore, the design cannot be fully verified until simulation is run on the gate-level design.

The following functionality might require component instantiation or functional implication:

- Clock-gating circuitry for power savings
- Asynchronous logic with potential hazards

This functionality includes asynchronous logic and asynchronous signals that are valid during certain states.

- Data-path circuitry

This functionality includes large multiplexers; instantiated wide banks of multiplexers; memory elements, such as RAM or ROM; and black box macro cells.

For more information about component implication, see the HDL Compiler documentation.

Guidelines for Modules

Observe these guidelines for modules:

- Avoid using logic expressions when you pass a value through ports.

The port list can include expressions, but expressions complicate debugging. In addition, isolating a problem related to the bit field is difficult, particularly if that bit field leads to internal port quantities that differ from external port quantities.

- Define local references as generics (VHDL) or parameters (Verilog). Do not pass generics or parameters into modules.

4

Working With Libraries

This chapter presents basic library information. Design Compiler uses technology, symbol, and synthetic or DesignWare libraries to implement synthesis and to display synthesis results graphically. You must know how to carry out a few simple library commands so that Design Compiler uses the library data correctly.

This chapter contains the following sections:

- [Selecting a Semiconductor Vendor](#)
- [Understanding the Library Requirements](#)
- [Specifying Libraries](#)
- [Loading Libraries](#)
- [Listing Libraries](#)
- [Reporting Library Contents](#)

- Specifying Library Objects
- Directing Library Cell Usage
- Removing Libraries From Memory
- Saving Libraries

Selecting a Semiconductor Vendor

One of the first things you must do when designing a chip is to select the semiconductor vendor and technology you want to use. Consider the following issues during the selection process:

- Maximum frequency of operation
- Physical restrictions
- Power restrictions
- Packaging restrictions
- Clock tree implementation
- Floorplanning
- Back-annotation support
- Design support for libraries, megacells, and RAMs
- Available cores
- Available test methods and scan styles

Understanding the Library Requirements

Design Compiler uses these libraries:

- Technology libraries
- Symbol libraries
- DesignWare libraries

This section describes these libraries.

Technology Libraries

Technology libraries contain information about the characteristics and functions of each cell provided in a semiconductor vendor's library. Semiconductor vendors maintain and distribute the technology libraries.

Cell characteristics include information such as cell names, pin names, area, delay arcs, and pin loading. The technology library also defines the conditions that must be met for a functional design (for example, the maximum transition time for nets). These conditions are called design rule constraints.

In addition to cell information and design rule constraints, technology libraries specify the operating conditions and wire load models specific to that technology.

Design Compiler requires the technology libraries to be in .db format. In most cases, your semiconductor vendor provides you with .db formatted libraries. If you are provided with only library source code, see the Library Compiler documentation for information about generating technology libraries.

Design Compiler uses technology libraries for these purposes:

- Implementing the design function

The technology libraries that Design Compiler maps to during optimization are called target libraries. The target libraries contain the cells used to generate the netlist and definitions for the design's operating conditions.

The target libraries that are used to compile or translate a design become the local link libraries for the design. Design Compiler saves this information in the design's `local_link_library` attribute. For information about attributes, see [“Working With Attributes” on page 5-50](#).

- Resolving cell references

The technology libraries that Design Compiler uses to resolve cell references are called link libraries. Design Compiler uses the `link` command to resolve references in a design. The `link` command uses the `link_library` and `search_path` system variables.

In addition to technology libraries, link libraries can also include design files. The link libraries contain the descriptions of cells (library cells as well as subdesigns) in a mapped netlist.

Link libraries include both local link libraries (`local_link_library` attribute) and system link libraries (`link_library` variable).

For more information about resolving references, see [“Linking Designs” on page 5-14](#).

- Calculating timing values and path delays

The link libraries define the delay models that are used to calculate timing values and path delays. For information about the various delay models, see the Library Compiler documentation.

- Calculating power consumed

For information about calculating power consumption, see the *Power Compiler Reference Manual*.

Symbol Libraries

Symbol libraries contain definitions of the graphic symbols that represent library cells in the design schematics. Semiconductor vendors maintain and distribute the symbol libraries.

Design Compiler uses symbol libraries to generate the design schematic. You must use Design Vision to view the design schematic.

When you generate the design schematic, Design Compiler performs a one-to-one mapping of cells in the netlist to cells in the symbol library.

DesignWare Libraries

A DesignWare library is a collection of reusable circuit-design building blocks (components) that are tightly integrated into the Synopsys synthesis environment.

DesignWare components that implement many of the built-in HDL operators are provided by Synopsys. These operators include +, -, *, <, >, <=, >=, and the operations defined by if and case statements.

You can develop additional DesignWare libraries at your site by using DesignWare Developer, or you can license DesignWare libraries from Synopsys or from third parties. To use licensed DesignWare components, you need a license key.

Specifying Libraries

You use `dc_shell` variables to specify the libraries used by Design Compiler. [Table 4-1](#) lists the variables for each library type as well as the typical file extension for the library.

Table 4-1 Library Variables

Library type	Variable	Default	File extension
Target library	<code>target_library</code>	<code>{"your_library.db"}</code>	<code>.db</code>
Link library	<code>link_library</code>	<code>{"*", "your_library.db"}</code>	<code>.db</code>
Symbol library	<code>symbol_library</code>	<code>{"your_library.sdb"}</code>	<code>.sdb</code>
DesignWare library	<code>synthetic_library</code>	<code>{}</code>	<code>.sldb</code>

Specifying a Library Search Path

You can specify the library location by using either the complete path or only the file name. If you specify only the file name, Design Compiler uses the search path defined in the `search_path` variable to locate the library files. By default, the search path includes the current working directory and `$SYNOPSYS/libraries/syn`. Design Compiler looks for the library files, starting with the leftmost directory specified in the `search_path` variable, and uses the first matching library file it finds.

For example, assume that you have technology libraries named `my_lib.db` in both the `lib` directory and the `vhdl` directory. If the search path contains (in order) the `lib` directory, the `vhdl` directory, and the default search path, Design Compiler uses the `my_lib.db` file found in the `lib` directory, because it encounters the `lib` directory first.

You can use the `which` command to see which library files Design Compiler finds (in order).

```
dc_shell> which my_lib.db
{"/usr/lib/my_lib.db", "/usr/vhdl/my_lib.db"}
```

Specifying Technology Libraries

To specify technology libraries, you must specify the target library and link library.

Design Compiler uses the target library to build a circuit. During mapping, Design Compiler selects functionally correct gates from the target library. It also calculates the timing of the circuit, using the vendor-supplied timing data for these gates.

You use the `target_library` variable to specify the target library.

The syntax for `dcsh` mode is

```
target_library = my_tech.db
```

The syntax for `dctcl` mode is

```
set target_library my_tech.db
```

Design Compiler uses the link library to resolve design references. You use the `link_library` variable to specify a list of libraries that Design Compiler can use to resolve design references.

The syntax for `dcsh` mode is

```
link_library = {* my_tech.db}
```

The syntax for `dcctl` mode is

```
set link_library {* my_tech.db}
```

Note that you specify the same value for the target library and the link library (except when you are performing technology translation). For the link library, you should also specify the asterisk character (*), which makes Design Compiler also search the designs in memory when it is resolving cell references. If the `link_library` variable has no asterisk, the designs loaded in memory are not searched. As a result, designs might not be found during linking and might be unresolved.

When Design Compiler attempts to locate a library file, it first searches the memory. Next, it searches the library files specified in the `link_library` variable and lastly, it searches the paths defined in the `search_path` variable.

When you specify the files in the `link_library` variable, consider that Design Compiler searches these files from left to right when it resolves references, and it stops searching when it finds a reference. If you specify the link library as `{"*" lsi_10k.db}`, the designs in memory are searched before the `lsi_10k` library.

For more information about resolving references, see [“Linking Designs” on page 5-14](#).

Note that Design Compiler uses the first technology library found in the `link_library` variable as the main library. Design Compiler uses the main library to obtain default values and settings used in the absence of explicit specifications for operating conditions, wire load

selection group, wire load mode, and net delay calculation. Design Compiler obtains the following default values and settings from the main library:

- Unit definitions
- Operating conditions
- K-factors
- Wire load model selection
- Input and output voltage
- Timing ranges
- RC slew trip points
- Net transition time degradation tables

Note that if other libraries have units different from the main library units, Design Compiler converts all units to those that the main library uses.

Specifying DesignWare Libraries

You do not need to specify the standard synthetic library, `standard.sldb`, that implements the built-in HDL operators. The software 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).

For more information about using DesignWare libraries, see the *DesignWare User Guide*.

Loading Libraries

Design Compiler uses binary libraries (.db format for technology libraries and .sdb format for symbol libraries) and automatically loads these libraries when needed.

If your library is not in the appropriate binary format, use the `read_lib` command to compile the library source. The `read_lib` command requires a Library-Compiler license.

To manually load a binary library, use the `read_file` command.

```
dc_shell> read_file my_lib.db
dc_shell> read_file my_lib.sdb
```

Listing Libraries

Design Compiler refers to a library loaded in memory by its name. The library statement in the library source defines the library name.

To list the names of the libraries loaded in memory, use the `list_libs` command.

```
dc_shell> list_libs
my_lib      my_symbol_lib
```

To list the path and file name information along with the names, use the `list -libraries` command (dcsh mode only).

```
dc_shell> list -libraries
Library      File      Path
-----
my_lib       my_lib.db  /synopsys/libraries
my_symbol_lib my_lib.sdb /synopsys/libraries
```

Reporting Library Contents

Use the `report_lib` command to report the contents of a library. The `report_lib` command can report the following data:

- Library units
- Operating conditions
- Wire load models
- Cells (including cell exclusions, preferences, and other attributes)

Specifying Library Objects

Library objects are the vendor-specific cells and their pins.

The Design Compiler naming convention for library objects is

`[file:]library/cell[/pin]`

file

The file name of a technology library followed by a colon (:). If you have multiple libraries loaded in memory with the same name, you must specify the file name.

library

The name of a library in memory, followed by a slash (/).

cell

The name of a library cell.

pin

The name of a cell's pin.

For example, to set the `dont_use` attribute on the AND4 cell in the `my_lib` library, enter

```
dc_shell> set_dont_use my_lib/AND4
```

To set the `disable_timing` attribute on the Z pin of the AND4 cell in the `my_lib` library, enter one of the following commands (depending on your shell mode):

```
dc_shell> set_disable_timing find(pin, my_lib/AND4/Z)
```

```
dc_shell-t> set_disable_timing [get_pins my_lib/AND4/Z]
```

Directing Library Cell Usage

When Design Compiler maps a design to a technology library, it selects components (library cells) from that library. You can influence the choice of components (library cells) by

- Excluding cells from the target library
- Specifying cell preferences

Excluding Cells From the Target Library

Use the `set_dont_use` command to exclude cells from the target library. Design Compiler does not use these excluded cells during optimization.

This command affects only the copy of the library that is currently loaded into memory and has no effect on the version that exists on disk. However, if you save the library, the exclusions are saved and the cells are permanently disabled.

For example, to prevent Design Compiler from using the high-drive inverter INV_HD, enter

```
dc_shell> set_dont_use MY_LIB/INV_HD  
Performing set_dont_use on library cell 'MY_LIB/INV_HD'.  
1
```

Use the `remove_attribute` command to reinclude excluded cells in the target library.

```
dc_shell> remove_attribute MY_LIB/INV_HD dont_use  
Performing remove_attribute on library cell 'MY_LIB/INV_HD'.  
1
```

Specifying Cell Preferences

Use the `set_prefer` command to indicate preferred cells. You can issue this command with or without the `-min` option.

Use the command without the `-min` option if you want Design Compiler to prefer certain cells during the initial mapping of the design.

- Set the preferred attribute on particular cells to override the default cell identified by the library analysis step. This step occurs at the start of compilation to identify the starting cell size for the initial mapping.
- Set the preferred attribute on cells if you know the preferred starting size of the complex cells or the cells with complex timing arcs (such as memories and banked components).

You do not normally need to set the preferred attribute as part of your regular compile methodology because a good starting cell is automatically determined during the library analysis step.

Because nonpreferred gates can be chosen to meet optimization constraints, the effect of preferred attributes might not be noticeable after optimization.

For example, to set a preference for the low-drive inverter INV_LD, enter

```
dc_shell> set_prefer MY_LIB/INV_LD  
Performing set_prefer on library cell 'MY_LIB/INV_LD'.  
1
```

Use the `remove_attribute` command to remove cell preferences.

```
dc_shell> remove_attribute MY_LIB/INV_LD preferred  
Performing remove_attribute on library cell 'MY_LIB/INV_LD'.  
1
```

Use the `-min` option if you want Design Compiler to prefer fewer (but larger-area) buffers or inverters when it fixes hold-time violations. Normally, Design Compiler gives preference to smaller cell area over the number of cells used in a chain of buffers or inverters. You can change this preference by using the `-min` option, which tells Design Compiler to minimize the number of buffers or inverters by using larger area cells.

For example, to set a `hold_preferred` attribute for the inverter IV, enter

```
dc_shell> set_prefer -min class/IV  
Performing set_prefer on library cell 'class/IV'.  
1
```

Use the `remove_attribute` command to remove the `hold_preferred` cell attribute.

```
dc_shell> remove_attribute class/IV hold_preferred  
Performing remove_attribute on library cell 'class/IV'.  
{"class/IV"}
```

Removing Libraries From Memory

The `remove_design` command removes libraries from `dc_shell` memory. If you have multiple libraries with the same name loaded into memory, you must specify the path as well as the library name. Use the `list -libraries` command to see the path for each library in memory (dcsh mode only).

Saving Libraries

The `write_lib` command saves (writes to disk) a compiled library in Synopsys database, EDIF, or VHDL format.

5

Working With Designs in Memory

Design Compiler reads designs into memory from design files. Many designs can be in memory at any time. After a design is read in, you can change it in numerous ways, such as grouping or ungrouping its subdesigns or changing subdesign references.

This chapter contains the following sections:

- [Design Terminology](#)
- [Reading Designs](#)
- [Listing Designs in Memory](#)
- [Setting the Current Design](#)
- [Linking Designs](#)
- [Listing Design Objects](#)
- [Specifying Design Objects](#)

- Creating Designs
- Copying Designs
- Renaming Designs
- Changing the Design Hierarchy
- Editing Designs
- Translating Designs From One Technology to Another
- Removing Designs From Memory
- Saving Designs
- Working With Attributes

Design Terminology

Different companies use different terminology for designs and their components. This section describes the terminology used in the Synopsys synthesis tools.

About Designs

Designs are circuit descriptions that perform logical functions. Designs are described in various design formats, such as VHDL, Verilog HDL, state machine, and EDIF.

Logic-level designs are represented as sets of Boolean equations. Gate-level designs, such as netlists, are represented as interconnected cells.

Designs can exist and be compiled independently of one another, or they can be used as subdesigns in larger designs. Designs are flat or hierarchical.

Flat Designs

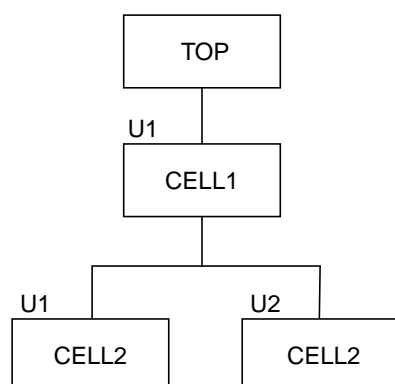
Flat designs contain no subdesigns and have only one structural level. They contain only library cells.

Hierarchical Designs

A hierarchical design contains one or more designs as subdesigns. Each subdesign can further contain subdesigns, creating multiple levels of design hierarchy. Designs that contain subdesigns are called parent designs.

[Figure 5-1](#) shows the three levels of hierarchy in the TOP design.

Figure 5-1 TOP Design Hierarchy



Design Objects

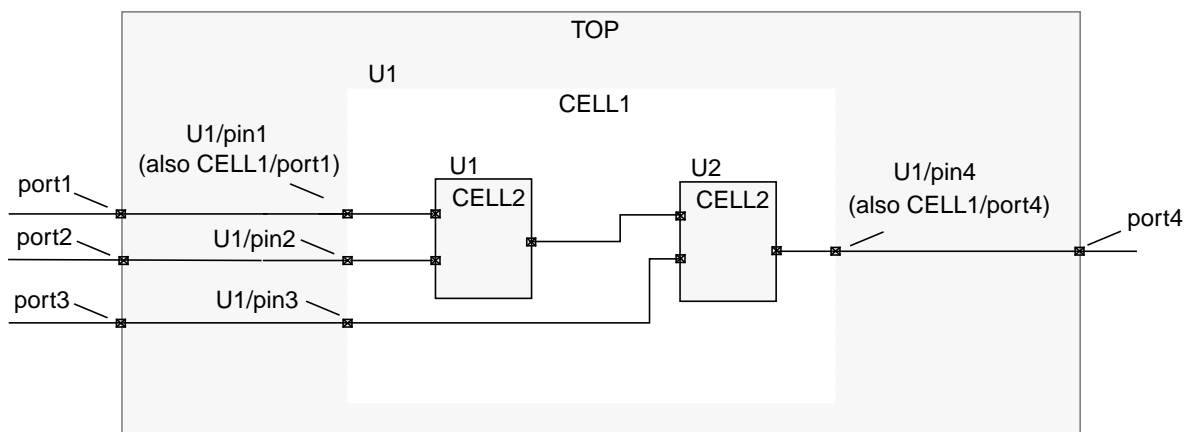
A design consists of cells, nets, ports, and pins. It can contain subdesigns and references to subdesigns and library cells.

Synopsys commands, attributes, and constraints are directed toward a design object.

Figure 5-2 shows the design objects in the TOP design example.

TOP has three input ports and one output port. The ports of CELL1 and CELL2 are also pins of TOP. TOP and CELL1 are parent designs. CELL2 is instantiated twice in CELL1.

Figure 5-2 Design Objects in TOP Design



Current Design

The active design (the design being worked on) is called the current design. Most commands are specific to the current design, that is, they operate within the context of the current design.

Cells, Instances, and References

A unique instance of a design within another design is called a hierarchical cell. A unique instance of a library cell within a design is called a leaf cell. The term *instance* is also used to refer to a cell.

Some commands work within the context of a hierarchical instance of the current design. The current instance defines the active instance for these instance-specific commands.

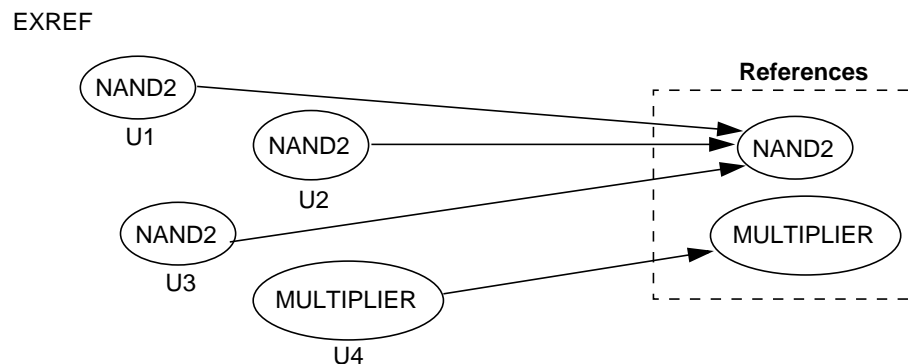
Cells are defined by their attributes, such as their functions, wiring (connections), size, and timing.

A design can contain multiple instances of identical subdesigns or library cells. The instantiated subdesign or library cell is called the reference. Each instance points to the same reference but has a unique name to differentiate it from the other instances.

References enable you to optimize every cell (such as a NAND gate) in a single design without affecting cells in other designs. The cell references in one design are independent of the same cell references in a different design.

Figure 5-3 shows the relationships among designs, cells, and references.

Figure 5-3 Cells and Cell References



The EXREF design contains two references: NAND2 and MULTIPLIER. NAND2 is instantiated three times, and MULTIPLIER is instantiated once.

The names given to the three instances of NAND2 are U1, U2, and U3. The references of NAND2 and MULTIPLIER in the EXREF design are independent of the same references in different designs.

For information about resolving references, see [“Linking Designs” on page 5-14](#).

Nets

Nets are the wires that connect ports to pins and pins to each other.

Ports

Ports are the inputs and outputs of a design. Port direction is designated as input, output, or inout.

Pins

Pins are the input and output of cells within a design (such as gates and flip-flops). The ports of a subdesign are pins within the parent design.

Reading Designs

[Table 5-1](#) lists the design file formats supported by Design Compiler. Design Compiler supports input and output of all formats listed in this table. All formats except .db, EDIF, equation, PLA, and state table require special license keys.

Table 5-1 Supported Design File Formats

Format	Description	Keyword	Extension
.db	Synopsys internal database format	db	.db
EDIF	Electronic Design Interchange Format (see the <i>EDIF 2 0 0 Interface User Guide</i>)	edif	.edif
equation	Synopsys equation format	equation	.eqn
LSI	LSI Logic Corporation (NDL) netlist format	lsi	.NET
Mentor	Mentor NETED do Format (output only)	mentor	.neted
MIF	Mentor Intermediate Format (input only)	mif	.mif

Table 5-1 *Supported Design File Formats (Continued)*

Format	Description	Keyword	Extension
PLA	Berkeley (Espresso) PLA format	pla	.pla
state table	Synopsys state table format	st	.st
TDL	Tegas Design Language netlist format	tdl	.tdl
Verilog	Verilog Hardware Description Language (see the HDL Compiler documentation)	verilog	.v
VHDL	VHSIC Hardware Description Language (see the HDL Compiler documentation)	vhdl	.vhd
XNF	Xilinx netlist format (see the <i>Design Compiler FPGA User Guide</i>)	xnf	.xnf

Design Compiler provides two ways to read design files:

- The `read_file` command

```
dc_shell> read_file -format keyword design_file
```

- The `analyze` and `elaborate` commands

```
dc_shell> analyze -format keyword design_file  
dc_shell> elaborate design_name
```

[Table 5-2](#) summarizes the differences between using the `read_file` command and using the `analyze` and `elaborate` commands to read design files.

Table 5-2 read_file Versus analyze and elaborate Commands

Comparison	read_file command	analyze and elaborate commands
Input formats	All formats	VHDL, Verilog
When to use	Netlists, precompiled designs, and so forth	Synthesize VHDL or Verilog
Generics	Cannot pass parameters (must use directives in HDL)	Allows you to set parameter values on the <code>elaborate</code> command line
Architecture	Cannot specify architecture to be elaborated	Allows you to specify architecture to be elaborated

A design file exists in your host computer's file system. When Design Compiler reads a design file, it is stored in memory in the Synopsys internal database (.db) format. The Design Compiler optimization process works only on the designs loaded in memory.

For designs in memory, Design Compiler uses the naming convention *path_name / design.db*. The `path_name` argument is the directory from which the original file was read, and the `design` argument is the name of the design. If you later read in a design that has the same file name, Design Compiler overwrites the original design. To prevent this, use the `-single_file` option with the `read_file` command.

Using a Search Path

You can specify the design file location by using the complete path or only the file name. If you specify only the file name, Design Compiler uses the search path defined in the `search_path` variable to locate the design files. Design Compiler looks for the

design files starting with the leftmost directory specified in the `search_path` variable and uses the first design file it finds. When you specify the path, Design Compiler does not use the search path.

To see where Design Compiler finds a file when using the search path, use the `which` command. For example, enter

```
dc_shell> which my_design.db  
{/usr/designers/example/my_design.db}
```

Reading .db Files

The version of a .db file is the version of Design Compiler that created the file. For a .db file to be read into Design Compiler, its file version must be the same as or earlier than the version of Design Compiler you are running.

If you attempt to read in a .db file generated by a Design Compiler version that is later than the Design Compiler version you are using, an error message appears. The error message provides details about the version mismatch.

Reading HDL Designs

Use the following process to read HDL designs:

1. Analyze the top-level design and all subdesigns in bottom-up order (to satisfy any dependencies).
2. Elaborate the top-level design and any subdesigns that require parameters to be assigned or overwritten.

Analyzing Designs

The `analyze` command

- Reads an HDL source file
- Checks it for errors (without building generic logic for the design)
- Creates HDL library objects in an HDL-independent intermediate format
- Stores the intermediate files in a location you define

If the `analyze` command reports errors, fix them in the HDL source file and run `analyze` again.

Once a design is analyzed, you must reanalyze the design only when you change it.

Elaborating Designs

The `elaborate` command creates a technology-independent design from the intermediate files produced during analysis. You can override default parameter values during elaboration. Elaboration replaces the HDL arithmetic operators in the code with DesignWare components and determines the correct bus size.

For more information about the `analyze` and `elaborate` commands, see the HDL Compiler documentation.

Listing Designs in Memory

To list the names of the designs loaded in memory, use the `list_designs` command.

```
dc_shell> list_designs
A (*)      B      C
1
```

The asterisk (*) next to design A shows that A is the current design.

To list the memory file name corresponding to each design name, use the `-show_file` option.

```
dc_shell> list_designs -show_file

/user1/designs/design_A/A.db
A (*)

/home/designer/dc/B.db
B      C
1
```

The asterisk (*) next to design A shows that A is the design you are working on. File B.db contains both designs B and C.

To check for duplicate designs loaded in memory, use the `list_duplicate_designs` command (dcsh mode only).

```
dc_shell> list_duplicate_designs
Warning: Multiple designs in memory with the same design
name.

  Design      File      Path
  -----
  seq2        A.db      /home/designer/dc
  seq2        B.db      /home/designer/dc
1
```

Setting the Current Design

The `current_design` variable points to the current design and is set in the following ways:

- With the `read_file` command

When the `read_file` command successfully finishes processing, it sets the current design to the design that was read in.

```
dc_shell> read_file -format edif MY_DESIGN.edif
Loading edif file '/designs/ex/MY_DESIGN.edif'
Current design is now '/designs/ex/
MY_DESIGN.edif:MY_DESIGN'
{"MY_DESIGN"}
```

- With the `elaborate` command
- With the `current_design` command

Use this command to set any design in `dc_shell` memory as the current design.

```
dc_shell> current_design ANY_DESIGN
Current design is 'ANY_DESIGN'.
{"ANY_DESIGN"}
```

To display the name of the current design, enter one of the following commands (depending on your shell mode):

```
dc_shell> list current_design
current_design = "/usr/home/designs/
my_design.db:my_design"
1
```

```
dc_shell-t> printvar current_design
current_design = "/usr/home/designs/
my_design.db:my_design"
```

Linking Designs

For a design to be complete, it must connect to all the library components and designs it references. For each subdesign, there must be a reference that links the subdesign or component to the link libraries. This process is called linking the design or resolving references.

Design Compiler resolves references by carrying out the following steps:

1. It determines which library components and subdesigns are referenced in the current design and its hierarchy.
2. It searches the link libraries to locate these references.
3. It links the located references to the design.

Design Compiler first searches the libraries and design files defined in the current design's `local_link_library` attribute, then searches the libraries and design files defined in the `link_library` variable.

Note:

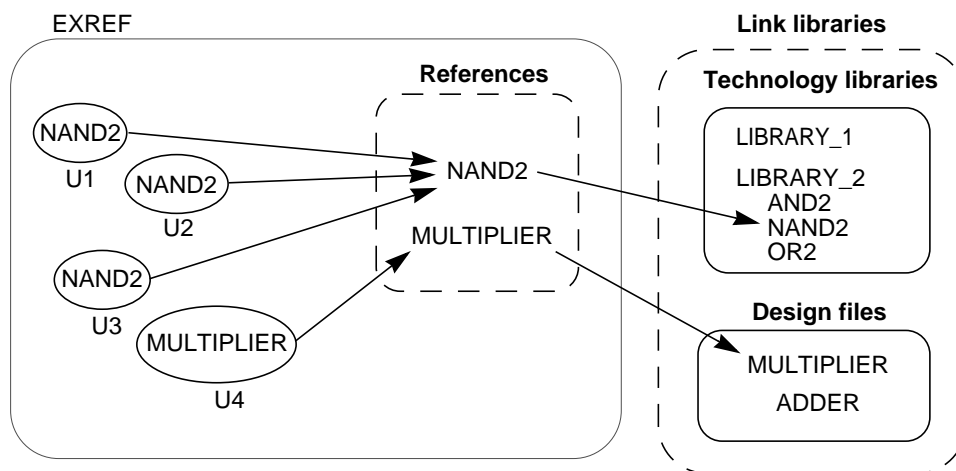
In a hierarchical design, Design Compiler considers only the top-level design's local link library. It ignores local link libraries associated with the subdesigns.

Design Compiler uses the first reference it locates. If it locates additional references with the same name, it generates a warning message identifying the ignored, duplicate references. If Design Compiler does not find the reference, a warning appears advising that the reference cannot be resolved.

By default, the case sensitivity of the linking process depends on the source of the references. To explicitly define the case sensitivity of the linking process, set the `link_force_case` variable.

The arrows in [Figure 5-4](#) show the connections that the linking process added between the cells, references, and link libraries. In this example, Design Compiler finds library component NAND2 in the LIBRARY_2 technology library; it finds subdesign MULTIPLIER in a design file.

Figure 5-4 Resolving References



You can link the design either manually or automatically.

Linking a Design Manually

Use the `link` command to manually link a design. The `link` command removes existing links before starting the link process.

Linking a Design Automatically

The following `dc_shell` commands automatically link designs:

- `compile`
- `create_schematic`
- `group`
- `check_design`
- `report_timing`, `report_constraints`, and other `report_*` commands
- `compare_design`

When Design Compiler links designs automatically, it does not remove existing links. The automatic link process works only on unlinked components and subdesigns.

Changing Design References

Use the `change_link` command to change the component or design to which a cell or reference is linked.

- For a cell, the link for that cell is changed.
- For a reference, the link is changed for all cells having that reference.

The link can be changed only to a component or design that has the same number of ports with the same size and direction as the original reference.

When you use `change_link`, all link information is copied from the old design to the new design. If the old design is a synthetic module, all attributes of the old synthetic module are moved to the new link. After using `change_link`, manually link the design with the `link` command.

Listing Design Objects

Design Compiler provides commands for accessing various design objects. These commands refer to design objects located in the current design. Each command performs one of the following actions:

- **List**
Provides a listing with minimal information.
- **Display**
Provides a report that includes characteristics of the design object.
- **Return**
Returns a list (dcsh mode) or collection (dctl mode) that can be used as input to another `dc_shell` command.

[Table 5-3](#) lists the commands and the actions they perform.

Table 5-3 Commands to Access Design Objects

Object	Command	Action
Instance	<code>list_instances</code>	Lists instances and their references.
	<code>report_cell</code>	Displays information about instances.

Table 5-3 Commands to Access Design Objects (Continued)

Object	Command	Action
Reference	<code>report_reference</code>	Displays information about references.
Port	<code>report_port</code> <code>report_bus</code> <code>all_inputs</code> <code>all_outputs</code>	Displays information about ports. Displays information about bused ports. Returns all input ports. Returns all output ports.
Net	<code>report_net</code> <code>report_bus</code>	Displays information about nets. Displays information about bused nets.
Clock	<code>report_clock</code> <code>all_clocks</code>	Displays information about clocks. Returns all clocks.
Register	<code>all_registers</code>	Returns all registers.

Note:

In `dctcl` mode, you can also use the `get_*` commands to create and list collections of cells, designs, libraries, library cells, library cell pins, nets, pins, and ports.

Specifying Design Objects

You can specify design objects by using either a relative path or an absolute path.

Using a Relative Path

If you use a relative path to specify a design object, the object must be in the current design. Specify the path relative to the current instance. The current instance is the frame of reference within the

current design. By default, the current instance is the top level of the current design. Use the `current_instance` command to change the current instance.

For example, to place a `dont_touch` attribute on hierarchical cell U1/U15 in the Count_16 design, you can enter either

```
dc_shell> current_design Count_16
Current design is 'Count_16'.
{"Count_16"}
dc_shell> set_dont_touch U1/U15
```

or

```
dc_shell> current_design Count_16
Current design is 'Count_16'.
{"Count_16"}
dc_shell> current_instance U1
Current instance is '/Count_16/U1'.
"/Count_16/U1"
dc_shell> set_dont_touch U15
```

In the first command sequence, the frame of reference remains at the top level of design Count_16. In the second command sequence, the frame of reference changes to instance U1. Design Compiler interprets all future object specifications relative to instance U1.

To reset the current instance to the top level of the current design, enter the `current_instance` command without an argument.

```
dc_shell> current_instance
```

The `current_instance` variable points to the current instance. To display the current instance, enter one of the following commands (depending on your shell mode):

```
dc_shell> list current_instance
```

```
current_instance = "Count_16/U1"  
1
```

```
dc_shell-t> printvar current_instance  
current_instance = "Count_16/U1"
```

In dcsh mode, the `current_reference` variable points to the reference of the current instance. This variable is not supported in dctcl mode. To display the reference of the current instance, enter the following command (dcsh mode only):

```
dc_shell> list current_reference  
current_reference = "/usr/designs/Count_16.db:Count_4"  
1
```

Using an Absolute Path

When you use an absolute path to specify a design object, the object can be in any design in dc_shell memory. Use the following syntax to specify an object by using an absolute path:

[file:]design/object

file

The path name of a memory file followed by a colon (:). Use the file argument when multiple designs in memory have the same name.

design

The name of a design in dc_shell memory.

object

The name of the design object, including its hierarchical path. If several objects of different types have the same name and you do not specify the object type, Design Compiler looks for the object by using the types allowed by the command.

To specify an object type, use either the `find` command in `dcsh` mode or the `get_*` command in `dctl` mode. For more information about these commands, see the *Design Compiler Command-Line Interface Guide*.

For example, to place a `dont_touch` attribute on hierarchical cell U1/U15 in the `Count_16` design, enter

```
dc_shell> set_dont_touch \  
          /usr/designs/Count_16.db:Count_16/U1/U5
```

Creating Designs

The `create_design` command creates a new design. The memory file name is *my_design.db*, and the path is the current working directory.

```
dc_shell> create_design my_design
Creating design 'my_design' in file 'my_design.db'.
1
dc_shell> list_designs -show_file

/designs/A.db
A (*)

/designs/B.db
B

/usr/work/my_design.db
my_design
1
```

Designs created with `create_design` contain no design objects. Use the appropriate create commands (such as `create_clock`, `create_cell`, or `create_port`) to add design objects to the new design. For information about these commands, see [“Editing Designs” on page 5-38](#).

Copying Designs

The `copy_design` command copies a design in memory and renames the copy. The new design has the same path and memory file as the original design.

```
dc_shell> copy_design A A_NEW
Copying design 'A' to 'A_NEW'
1
dc_shell> list_designs -show_file

/designs/A.db
A      A_NEW

/designs/B.db
B
1
```

You can use the `copy_design` command with the `change_link` command to manually create unique instances. For example, assume that a design has two identical cells, U1 and U2, both linked to COMP.

Enter the following commands to create unique instances:

```
dc_shell> copy_design COMP COMP1
Copying design 'COMP' to 'COMP1'
1
dc_shell> change_link U1 COMP1
Performing change_link on cell 'U1'.
1
dc_shell> copy_design COMP COMP2
Copying design 'COMP' to 'COMP2'
1
dc_shell> change_link U2 COMP2
Performing change_link on cell 'U2'.
1
```

Renaming Designs

The `rename_design` command renames a design in memory.

```
dc_shell> list_designs -show_file

/designs/X.db
A      B
1
dc_shell> rename_design A A_NEW
Moving design 'A' to 'A_NEW'
1
dc_shell> list_designs -show_file

/designs/X.db
A_NEW  B
1
```

Note:

Renaming designs might cause unresolved references during linking.

Changing the Design Hierarchy

When possible, reflect the design partitioning in your HDL description. If your HDL code is already developed, Design Compiler enables you to change the hierarchy without modifying the HDL description.

The `report_hierarchy` command displays the design hierarchy. Use this command to understand the current hierarchy before making changes and to verify the hierarchy changes.

Design Compiler provides the following hierarchy manipulation capabilities:

- Adding levels of hierarchy
- Removing levels of hierarchy
- Merging cells from different subdesigns

The following sections describe these capabilities.

Adding Levels of Hierarchy

Adding a level of hierarchy is called grouping. You can create a level of hierarchy by grouping cells or related components into subdesigns.

Grouping Cells Into Subdesigns

The `group` command groups cells (instances) in the design into a new subdesign, creating a new level of hierarchy. The grouped cells are replaced by a new instance (cell) that references the new subdesign.

The ports of the new subdesign are named after the nets to which they are connected in the design. The direction of each port of the new subdesign is determined from the pins of the corresponding net.

To create a new subdesign by using the `group` command, specify the following arguments and options in the command line:

- Specify the cells included in the new subdesign as the command-line argument.

All cells must be children of the current instance. You can exclude cells from the specified list by using the `-except` option.

- Specify the name of the new subdesign, using the `-design_name` option.
- Specify the new instance name, using the `-cell_name` option (optional).

If you do not specify an instance name, Design Compiler creates one for you. The created instance name has the format `Un`, where *n* is an unused cell number (for example, `U107`).

For example, to group three cells into a new design named `sample`, enter

```
dc_shell> group {cell1 cell2 cell3} -design_name sample
```

To group all cells that begin with alu into a new design uP with cell name UCELL, enter

```
dc_shell> group "alu*" -design_name uP -cell_name UCELL
```

Note:

Grouping cells might not preserve all the attributes and constraints of the original cells.

Grouping Related Components Into Subdesigns

You also use the `group` command (but with different options) to group related components into subdesigns.

To group related components,

- Specify the component type, using one of the options shown in [Table 5-4](#).

Table 5-4 Component Grouping Options

Component	Options
Bused gates	-hdl_bussed
Combinational logic	-logic
Finite state machines	-fsm
HDL blocks	-hdl_all_blocks -hdl_block <i>block_name</i>
PLA specifications	-pla

- Specify the name of the new subdesign, using the `-design_name` option.

- Specify the new instance name, using the `-cell_name` option (optional).

If you do not specify an instance name, Design Compiler creates one for you. The created instance name has the format `Un`, where *n* is an unused cell number (for example, `U107`).

For example, to group all cells in the HDL function bar in the process `ftj` into design `new_block`, enter

```
dc_shell> group -hdl_block ftj/bar -design_name new_block
```

To group all bused gates beneath process `ftj` into separate levels of hierarchy, enter

```
dc_shell> group -hdl_block ftj -hdl_bussed
```

Removing Levels of Hierarchy

Design Compiler does not optimize across hierarchical boundaries; therefore, you might want to remove the hierarchy within certain designs. By doing so, you might be able to improve timing results.

Removing a level of hierarchy is called ungrouping. Ungrouping merges subdesigns of a given level of the hierarchy into the parent cell or design. Ungrouping can be done before optimization or during optimization (either explicitly or automatically).

Note:

Designs, subdesigns, and cells that have the `dont_touch` attribute cannot be ungrouped (including auto-ungrouping) before or during optimization.

Ungrouping Hierarchies Before Optimization

You use the `ungroup` command to ungroup one or more designs before optimization.

To ungroup a design before compile, enter the `ungroup` command on the command line and follow these steps:

1. Use the `-start_level` option or `-flatten` option to ungroup cells recursively.

By default, the `ungroup` command ungroups only one level of hierarchy in each cell.

To ungroup cells recursively starting at any hierarchical level below the current design, use the `-start_level` option. You must specify an argument for this option: 0 refers to the current design; 1, one level below the current design; 2, two levels below the current design, and so on. Note that cells that are at the level specified by the `-start_level` option are included in the ungrouping.

To ungroup each cell recursively until all levels of hierarchy within the current design (instance) are removed, specify the `-flatten` option.

Note:

You cannot use `-flatten` or `-small` with the `-start_level` option. The value that you assign to the `-start_level` option cannot be 0, that is, you cannot ungroup the current design.

2. Specify the cells to be ungrouped.

Provide a list of cells to be ungrouped, or use the `-all` option to ungroup all cells in the current design. Note that with this option Design Compiler does not ungroup cells recursively.

When you provide a list of cells to be ungrouped, you can specify only those cells that are the immediate children of the current design.

When you use the `-all` option with the `-start_level` option, Design Compiler recursively ungroups all cells starting at the specified hierarchical level.

3. Specify the prefix to be used to name the cells to be ungrouped.

If you do not specify a prefix, Design Compiler uses the prefix *old_cell_name/*. When you use the `-flatten` option, do not specify a prefix, because the default cell names are more descriptive than cell names generated with a specified prefix.

If the specified or default prefix creates a nonunique name, Design Compiler adds a number to the end of the cell name to make it unique.

The following examples illustrate how to use the `ungroup` command:

- To ungroup several cells, enter

```
dc_shell> ungroup {high_decoder_cell low_decoder_cell}
```

- To ungroup the cell U1 and specify the prefix to use when creating new cells, enter

```
dc_shell> ungroup U1 -prefix "U1_"
```

- To completely flatten the current design, enter

```
dc_shell> ungroup -all -flatten
```

- To recursively ungroup cells belonging to CELL_X, which is three hierarchical levels below the current design, enter

```
dc_shell> ungroup -start_level 3 CELL_X
```

- To recursively ungroup cells that are three hierarchical levels below the current design and belong to cells U1 and U2 (U1 and U2 are child cells of the current design), enter

```
dc_shell> ungroup -start_level 2 {U1 U2}
```

- To recursively ungroup all cells that are three hierarchical levels below the current design, enter

```
dc_shell> ungroup -start_level 3 -all
```

Note:

If you ungroup cells and then use the `change_names` command to modify the hierarchy separator (/), you might lose attribute and constraint information.

Ungrouping Hierarchies During Optimization

You can ungroup designs during optimization either explicitly or automatically.

Ungrouping Hierarchies Explicitly During Optimization. You can control which designs are ungrouped during optimization by using the `set_ungroup` command followed by the `compile` command or the `-ungroup_all` compile option.

- Use the `set_ungroup` command when you want to specify the cells or designs to be ungrouped. This command assigns the `ungroup` attribute to the specified cells or referenced designs. If you set the attribute on a design, all cells that reference the design are ungrouped.

For example, to ungroup cell U1 during optimization, enter the following commands:

```
dc_shell> set_ungroup U1  
dc_shell> compile
```

To see whether an object has the `ungroup` attribute set, use the `get_attribute` command.

```
dc_shell> get_attribute object ungroup
```

To remove an `ungroup` attribute, use the `remove_attribute` command or set the `ungroup` attribute to `false`.

```
dc_shell> set_ungroup object false
```

- Use the `-ungroup_all` compile option to remove all lower levels of the current design hierarchy (including DesignWare parts). For example, enter

```
dc_shell> compile -ungroup_all
```

Ungrouping Hierarchies Automatically During Optimization.

Design Compiler provides two options to automatically ungroup hierarchies: cell count-based auto-ungrouping and delay-based auto-ungrouping.

To use the auto-ungrouping capability, enter

```
compile -auto_ungroup area|delay
```

You can use only one argument at a time: either the `area` argument for cell count-based auto-ungrouping or the `delay` argument for delay-based auto-ungrouping.

Note:

You can use the auto-ungrouping capability for all `compile` options except `-top` and `-incremental_mapping`.

Before ungrouping begins, the tool issues a message to indicate that the specified hierarchy is being ungrouped.

After auto-ungrouping, use the `report_auto_ungroup` command to get a report on the hierarchies that were ungrouped during cell count-based auto-ungrouping or delay-based auto-ungrouping. This report gives instance names, cell names, and the number of instances for each ungrouped hierarchy.

Cell Count-Based Auto-Ungrouping

Cell count-based auto-ungrouping ungroups small hierarchies and is used essentially for area optimization. You use this `compile` option if you want to control explicitly when the `compile` command ungroups the small hierarchies in the current design and its subdesigns. You define the maximum size of the hierarchy to be ungrouped by setting the `compile_auto_ungroup_area_num_cells` `compile` variable. The default is 30. Note that the number-of-cells limit of a hierarchy refers to the number of child cells in that hierarchy (that is, the cells are not counted recursively).

Delay-Based Auto-Ungrouping

Delay-based auto-ungrouping ungroups hierarchies along the critical path and is used essentially for timing optimization.

You define the maximum size of the hierarchy to be ungrouped by setting the `compile_auto_ungroup_delay_num_cells` compile variable. The default is 500. The number-of-cells limit of a hierarchy refers to the number of child cells in that hierarchy (that is, the cells are not counted recursively).

Note that DesignWare components are not ungrouped because they are already highly optimized and significant improvements in area or timing are unlikely. In addition, if the design being ungrouped has no timing violations, the tool issues a message to indicate that delay-based auto-ungrouping will not be performed.

Delay-based auto-ungrouping attempts to improve the overall timing of the design by ungrouping those hierarchies that are most likely to benefit from the extra boundary optimizations that ungrouping provides. Such hierarchies contain paths that are either critical or likely to become critical after subsequent optimization steps. Delay-based auto-ungrouping thus offers a less CPU-intensive alternative to `-ungroup_all` for improving design timing.

Cases in Which Auto-Ungrouping Is Not Performed

For both types of auto-ungrouping, a hierarchy is not ungrouped in the following cases:

- The wire load model for the hierarchy is different from the wire load model of the parent hierarchy.

A warning message is issued to indicate that the user has specified different wire loads and the hierarchy cannot be ungrouped.

You can override this behavior by setting the `compile_autoungroup_override_wlm` variable to true (the default is false). The ungrouped child cells of the hierarchy then inherit the wire load model of the parent hierarchy. Consequently, the child cells might have a more pessimistic wire load model. To ensure that the cells that are ungrouped into different wire load models are updated with the correct delays, set the `auto_ungroup_preserve_constraints` variable to true (in addition to setting the `compile_autoungroup_override_wlm` variable to true).

- Constraints or timing exceptions are set on pins of the hierarchy.

You can override this behavior by setting the `auto_ungroup_preserve_constraints` variable to true. Design Compiler ungroups the hierarchy and moves timing constraints to adjacent, persistent pins, that is, pins on the same net that remain after ungrouping.

For more information on preserving timing constraints, see [“Preserving Hierarchical Pin Timing Constraints During Ungrouping” on page 5-35](#).

- The hierarchy has more child cells than that specified by `compile_auto_ungroup_area_num_cells` or `compile_auto_ungroup_delay_num_cells`.
- The hierarchy has a `dont_touch` attribute or `ungroup` attribute.

For more information on these compile variables and options, see the man pages.

Preserving Hierarchical Pin Timing Constraints During Ungrouping

Hierarchical pins are removed when a cell is ungrouped. Depending on whether you are ungrouping a hierarchy before optimization or after optimization, Design Compiler handles timing constraints placed on hierarchical pins in different ways. The table below summarizes the effect that ungrouping has on timing constraints within different compile flows.

Table 5-5 Preserving Hierarchical Pin Timing Constraints

Compile flow	Effect on hierarchical pin timing constraints
Ungrouping a hierarchy before optimization by using <code>ungroup</code> :	Timing constraints placed on hierarchical pins are preserved. In previous releases, timing attributes placed on the hierarchical pins of a cell were not preserved when that cell was ungrouped. If you want your current optimization results to be compatible with previous results, set the <code>ungroup_preserve_constraints</code> variable to false. The default for this variable is true, which specifies that timing constraints will be preserved.
Ungrouping a hierarchy during optimization by using <code>compile -ungroup_all</code> or <code>set_ungroup</code> followed by <code>compile</code> :	Timing constraints placed on hierarchical pins are not preserved. To preserve timing constraints, set the <code>auto_ungroup_preserve_constraints</code> variable to true.

Table 5-5 Preserving Hierarchical Pin Timing Constraints(Continued)

Compile flow	Effect on hierarchical pin timing constraints
<p>Automatically ungrouping a hierarchy during optimization, that is, using:</p> <p><code>compile -auto_ungroup\ area delay</code></p>	<p>Design Compiler does not ungroup the hierarchy.</p> <p>To make Design Compiler ungroup the hierarchy and preserve timing constraints, set the <code>auto_ungroup_preserve_constraints</code> variable to true.</p>

When preserving timing constraints, Design Compiler reassigns the timing constraints to appropriate adjacent, persistent pins (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.

When you apply ungrouping to an unmapped design, the constraints on a hierarchical pin are moved to a leaf cell and the `size_only` attribute is assigned. However, the constraints are preserved through the compile process only if there is a one-to-one match between the unmapped cell and a cell from the target library.

Only the timing constraints set with the following commands are preserved:

- `set_false_path`
- `set_multicycle_path`
- `set_min_delay`
- `set_max_delay`
- `set_input_delay`
- `set_output_delay`
- `set_disable_timing`
- `set_case_analysis`
- `create_clock`
- `create_generated_clock`
- `set_propagated_clock`
- `set_clock_latency`

Note:

The `set_rtl_load` constraint is not preserved. Also, only the timing constraints of the current design are preserved. Timing constraints in other designs might be lost as a result of ungrouping hierarchy in the current design.

Merging Cells From Different Subdesigns

To merge cells from different subdesigns into a new subdesign,

1. Group the cells into a new design.

2. Ungroup the new design.

For example, the following command sequence creates a new alu design that contains the cells that initially were in subdesigns u_add and u_mult.

```
dc_shell> group {u_add u_mult} -design alu
dc_shell> current_design alu
dc_shell> ungroup -all
dc_shell> current_design top_design
```

Editing Designs

Design Compiler provides commands for incrementally editing a design that is in memory. These commands allow you to change the netlist or edit designs by using dc_shell commands instead of an external format.

Table 5-6 Commands to Edit Designs

Object	Task	Command
Cells	Create a cell	create_cell
	Delete a cell	remove_cell
Nets	Create a net	create_net
	Connect a net	connect_net
	Disconnect a net	disconnect_net
	Delete a net	remove_net
Ports	Create a port	create_port
	Delete a port	remove_port
Buses	Create a bus	create_bus
	Delete a bus	remove_bus

When connecting or disconnecting nets, use the `all_connected` command to see the objects that are connected to a net, port, or pin.

For example, the following command sequences replace the reference for cell U8 with a high-power inverter.

Table 5-7 Design Editing Examples

dcsh Example	dctcl Example
<pre>dc_shell> find(pin, U8/*) {"U8/A", "U8/Z"}</pre>	<pre>dc_shell-t> get_pins U8/* {"U8/A", "U8/Z"}</pre>
<pre>dc_shell> all_connected U8/A {"n66"}</pre>	<pre>dc_shell-t> all_connected U8/A {"n66"}</pre>
<pre>dc_shell> all_connected U8/Z {"OUTBUS[10]"}</pre>	<pre>dc_shell-t> all_connected U8/Z {"OUTBUS[10]"}</pre>
<pre>dc_shell> remove_cell U8 Removing cell 'U8' in design 'top'. 1</pre>	<pre>dc_shell-t> remove_cell U8 Removing cell 'U8' in design 'top'. 1</pre>
<pre>dc_shell> create_cell U8 IVP Creating cell 'U8' in design 'top'. 1</pre>	<pre>dc_shell-t> create_cell U8 IVP Creating cell 'U8' in design 'top'. 1</pre>
<pre>dc_shell> connect_net n66 \ find(pin,U8/A) Connecting net 'n66' to pin 'U8/A'. 1</pre>	<pre>dc_shell-t> connect_net n66 \ [get_pins U8/A] Connecting net 'n66' to pin 'U8/A'. 1</pre>
<pre>dc_shell> connect_net OUTBUS[10] \ find(pin,U8/Z) Connecting net 'OUTBUS[10]' to pin 'U8/Z'. 1</pre>	<pre>dc_shell-t> connect_net OUTBUS[10] \ [get_pins U8/Z] Connecting net 'OUTBUS[10]' to pin 'U8/Z'. 1</pre>

Note:

You can also achieve the same result by using the `change_link` command instead of the series of commands listed in [Table 5-7](#). For example, the following command replaces the reference for cell U8 with a high-power inverter.

```
dc_shell> change_link U8 IVP
```

Translating Designs From One Technology to Another

The `translate` command translates a design from one technology to another.

Designs are translated cell by cell from the original technology library to a new technology library, preserving the gate structure of the original design. The translator uses the functional description of each existing cell (component) to determine the matching component in the new technology library (target library). If no exact replacement exists for a component, it is remapped with components from the target library.

You can influence the replacement-cell selection by preferring or disabling specific library cells (`set_prefer` and `set_dont_use` commands) and by specifying the types of registers (`set_register_type` command). The target libraries are specified in the `target_library` variable. The `local_link_library` variable of the top-level design is set to the `target_library` value after the design is linked.

The `translate` command does not operate on cells or designs having the `dont_touch` attribute. After the translation process, Design Compiler reports cells that are not successfully translated. During the verification phase, Design Compiler applies the `compare_design` script.

Procedure to Translate Designs

The following procedure works for most designs, but manual intervention might be necessary for some complex designs.

To translate a design,

1. Read in your mapped design.

```
dc_shell> read_file design.db
```

2. Set the target library to the new technology library.

```
dc_shell> target_library = { target_lib.db }
```

or

```
dc_shell-t> set target_library target_lib.db
```

3. Invoke the `translate` command.

```
dc_shell> translate
```

After a design is translated, you can optimize it (using the `compile` command) to improve the implementation in the new technology library.

Restrictions on Translating Between Technologies

Keep the following restrictions in mind when you translate a design from one technology to another:

- The `translate` command translates functionality logically but does not preserve drive strength during translation. It always uses the lowest drive strength version of a cell, which might produce a netlist with violations.
- When you translate CMOS three-state cells into FPGA, functional equivalents between the technologies might not exist.

- Buses driven by CMOS three-state components must be fully decoded (Design Compiler can assume that only one bus driver is ever active). If this is the case, bus drivers are translated into control logic. To enable this feature, set the `compile_assume_fully_decoded_three_state_buses` variable to true before translating.
- If a three-state bus within a design is connected to one or more output ports, translating the bus to a multiplexed signal changes the port functionality. Because `translate` does not change port functionality, this case is reported as a translation error.

Removing Designs From Memory

The `remove_design` command removes designs from `dc_shell` memory. For example, after completing a compilation session and saving the optimized design, you can use `remove_design` to delete the design from memory before reading in another design.

By default, the `remove_design` command removes only the specified design. To remove its subdesigns, specify the `-hierarchy` option. To remove all designs (and libraries) from memory, specify the `-all` option.

If you defined variables that reference design objects, Design Compiler removes these references when you remove the design from memory. This prevents future commands from attempting to operate on nonexistent design objects.

[Table 5-8](#) provides dcsh and dctcl examples that show the effects of the `remove_design` command.

Table 5-8 remove_design Examples

dcsh Example	dctcl Example
dc_shell> PORTS = all_inputs() {"A0", "A1", "A2", "A3"}	dc_shell-t> set PORTS [all_inputs] {"A0", "A1", "A2", "A3"}
dc_shell> list PORTS PORTS = {"A0", "A1", "A2", "A3"}	dc_shell-t> query_objects \$PORTS PORTS = {"A0", "A1", "A2", "A3"}
dc_shell> remove_design Removing design 'top' 1	dc_shell-t> remove_design Removing design 'top' 1
dc_shell> list PORTS PORTS = {}	dc_shell-t> query_objects \$PORTS Error: No such collection '_sel2' (SEL-001)

Saving Designs

You can save (write to disk) the designs and subdesigns of the design hierarchy at any time, using different names or formats. After a design is modified, you should manually save it. Design Compiler does not automatically save designs before it exits.

[Table 5-1 on page 5-7](#) lists the design file formats supported by Design Compiler. All formats except .db, EDIF, equation, PLA, and state table require special license keys.

The `write` command converts designs in memory to a format you specify and saves that representation to disk. By default, Design Compiler saves the current design in .db format to the file `./ design_name.db`.

```
dc_shell> write
```

To write a hierarchical design and its subdesigns, specify only the top-level design; do not specify all the design's files. By default, Design Compiler writes each design to a separate file, so to write out all subdesigns with the top-level design, you must use the `-hierarchy` option.

```
dc_shell> write -hierarchy top_design
```

To save the design or designs to a single output file, use the `-output` option to specify the output file.

```
dc_shell> write -output file_name design_list
dc_shell> write -output file_name -hierarchy
```

To save all modified designs to their default files in `.db` format, enter one of the following commands (depending on your shell mode):

```
dc_shell> write -modified find( design, "*" )

dc_shell-t> foreach_in_collection design [get_designs] {
    set name [get_object_name $design]
    current_design $name
    write -modified -output $name.db
}
```

Saving Designs To Other Formats

When writing to formats other than `.db`, consider the naming requirements of the target environment. You might have to perform one or more of the following tasks before saving the design:

- If the target environment has restrictions on the design object names, use the `change_names` command to modify the names.

- If the target environment has specific requirements for bus delimiters, set the `bus_naming_style` variable to meet those requirements.
- If the target environment requires schematics, use the `create_schematic` command to generate the schematics.

To output the design in another format, use the `-format` option to specify the format.

```
dc_shell> write -format output_format
```

Ensuring Name Consistency Between the Design Database and the Netlist

Before writing a netlist from within `dc_shell`, make sure that all net and port names conform to the naming conventions for your layout tool. Also ensure that you are using a consistent bus naming style.

Some ASIC and EDA vendors have a program that creates a `.synopsys_dc.setup` file that includes the appropriate commands to convert names to their conventions. If you need to change any net or port names, use the `define_name_rules` and `change_names` commands.

Naming Rules Section of the `.synopsys_dc.setup` File.

[Table 5-9](#) shows sample naming rules as created by a specific layout tool vendor. These naming rules do the following:

- Set the bus naming format to `bus_name_N` or `bus_namen_N`
- Limit object names to alphanumeric characters
- Change DesignWare cell names to valid names (changes “*cell*” to “U” and “*-return” to “RET”)

Your vendor might use different naming conventions. Check with your vendor to determine the naming conventions you need to follow.

Table 5-9 Naming Rules Section of .synopsys_dc.setup File

dcsh Syntax	dctcl Syntax
<pre>bus_naming_style = %s_%d define_name_rules simple_names \ -allowed "A-Za-z0-9_" \ -last_restricted "_" \ -first_restricted "_" \ -map { {{"*cell*", "U"}, \ {"*-return", "RET"}} }</pre>	<pre>set bus_naming_style %s_%d define_name_rules simple_names \ -allowed "A-Za-z0-9_" \ -last_restricted "_" \ -first_restricted "_" \ -map { {{"*cell*", "U"}, \ {"*-return", "RET"}} }</pre>

Using the define_name_rules -map Command. [Example 5-1](#) shows how to use the `-map` option with `define_name_rules` to avoid an error in the format of the string. If you do not follow this convention, an error appears.

Example 5-1 Using define_name_rules -map

```
define_name_rules naming_convention -map { {{string1, string2}} } -type cell
```

For example, to remove trailing underscores from cell names, enter

```
dc_shell> define_name_rules naming_convention -map { {{_$, ""}} } -type cell
```

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

Resolving Naming Problems in the Flow. You might encounter conflicts in naming conventions in design objects, input and output files, and tool sets. In the design .db file, you can have many design objects (such as ports, nets, cells, logic modules, and logic module pins), all with their own naming conventions. Furthermore, you might be using several input and output file formats (such as DEF, PDEF,

Verilog, and EDIF) in your flow. Each file format is different and has its own syntax definitions. Using tool sets from several vendors can introduce additional naming problems.

To resolve naming issues, use the `change_names` command to ensure that all the file names match. Correct naming eliminates name escaping or mismatch errors in your design.

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

Methodology for Resolving Naming Issues

To resolve naming issues, make the name changes in the design .db file before you write any files. Your initial flow is

1. Read in your design RTL and apply constraints.

No changes to your method need to be made here.

2. Compile the design to produce a gate-level description.

Compile or reoptimize your design as you normally would, using your standard set of scripts.

3. Apply name changes and resolve naming issues. Use the `change_names` command and its Verilog or VHDL switch before you write the design.

Important:

Always use the `change_names -rules -[verilog|vhdl] -hierarchy` command whenever you want to write out a Verilog or VHDL design, because naming in the design.db file is not Verilog or VHDL compliant. For example, enter

```
change_names -rules verilog -hierarchy
```

Important:

If you need to change the bus naming style, you must first define the new bus naming style, using the `define_name_rules` command, and identify the bus naming style of the current design before you use the `change_names` command to define the new bus naming style rule. For example, to change the bus naming style from `%s[%d]` to `%s_%d`, enter one of the following command sequences (depending on your shell mode):

```
dc_shell> define_name_rules new_bus_naming_style \
          -target_bus_naming_style "%s_%d"
dc_shell> bus_naming_style="%s[%d]"
dc_shell> change_names -rule new_bus_naming_style \
          -hierarchy

dc_shell-t> define_name_rules new_bus_naming_style \
          -target_bus_naming_style "%s_%d"
dc_shell-t> set bus_naming_style "%s[%d]"
dc_shell-t> change_names -rule new_bus_naming_style \
          -hierarchy
```

4. Write files to disk. Use the `write -format verilog` command.

Look for reported name changes, which indicate you need to repeat step 3 and refine your name rules.

5. If all the appropriate name changes have been made, your output files matches the design .db file. Enter the following commands and compare the output.

```
write -format verilog -hierarchy -output "consistent.v"
write -format db -hierarchy -output "consistent.db"
```

6. Write the files for third-party tools.

If you need more specific naming control, use the `define_name_rules` command. See [“Using the define_name_rules -map Command” on page 5-46](#).

Summary of Commands for Changing Names

[Table 5-10](#) summarizes commands for changing names.

Table 5-10 Summary of Commands for Changing Names

To do this	Use this
Change the names of ports, cells, and nets in a design to be Verilog or VHDL compliant.	<code>change_names</code>
Show effects of <code>change_names</code> without making the changes.	<code>report_names</code>
Define a set of rules for naming design objects. Name rules are used by <code>change_names</code> and <code>report_names</code> .	<code>define_name_rules</code>
List available name rules.	<code>report_name_rules</code>

The following special cases apply:

- Synopsys database (.db) format is the only output format that can have designs containing unmapped synthetic library cells.
- The EDIF, LSI, and Mentor formats require a mapped design.
- The equation format requires a combinational design.
- Schematics are ignored by equation, LSI, PLA, state table, TDL, Verilog, and VHDL formats.
- The Mentor format requires schematics.

Working With Attributes

Attributes describe logical, electrical, physical, and other properties of objects in the design database. An attribute is attached to a design object and is saved with the design database.

Design Compiler uses attributes on the following types of objects:

- Entire designs
- Design objects, such as clocks, nets, pins, and ports
- Design references and cell instances within a design
- Technology libraries, library cells, and cell pins

An attribute has a name, a type, and a value. Attributes can have the following types: string, numeric, or logical (Boolean).

Some attributes are predefined and are recognized by Design Compiler; other attributes are user-defined. Appendix C lists the predefined attributes.

Some attributes are read-only. Design Compiler sets these attribute values and you cannot change them. Other attributes are read/write. You can change these attribute values at any time.

Most attributes apply to one object type; for example, the `rise_drive` attribute applies only to input and inout ports. Some attributes apply to several object types; for example, the `dont_touch` attribute can apply to a net, cell, port, reference, or

design. You can get detailed information about the predefined attributes that apply to each object type by using the commands listed in [Table 5-11](#).

Table 5-11 Commands to Get Attribute Descriptions

Object type	Command
All	<code>man attributes</code>
Designs	<code>man design_attributes</code>
Cells	<code>man cell_attributes</code>
Clocks	<code>man clock_attributes</code>
Nets	<code>man net_attributes</code>
Pins	<code>man pin_attributes</code>
Ports	<code>man port_attributes</code>
Libraries	<code>man library_attributes</code>
Library cells	<code>man library_cell_attributes</code>
References	<code>man reference_attributes</code>

Setting Attribute Values

To set the value of an attribute, use one of the following:

- An attribute-specific command

Use an attribute-specific command to set the value of the command's associated attribute.

For example,

```
dc_shell> set_dont_touch U1
```

- The `set_attribute` command

Use this command to set the value of any attribute or to define a new attribute and set its value.

For example, to set the `flatten` attribute to false on the design named `top`, enter

```
dc_shell> set_attribute top flatten false
```

If an attribute applies to more than one object type, Design Compiler searches the database for the named object. For information about the search order, see [“The Object Search Order” on page 5-55](#).

When you set an attribute on a reference (subdesign or library cell), the attribute applies to all cells in the design with that reference. When you set an attribute on an instance (cell, net, or pin), the attribute overrides any attribute inherited from the instance’s reference.

Viewing Attribute Values

To see all attributes on an object, use the `report_attribute` command.

```
dc_shell> report_attribute -object obj_type
```

To see the value of a specific attribute on an object, use the `get_attribute` command.

For example, to get the value of the maximum fanout on port `OUT7`, enter

```
dc_shell> get_attribute OUT7 max_fanout
Performing get_attribute on port 'OUT7'.
{3.000000}
```


If an attribute applies to more than one object type, Design Compiler searches the database for the named object. For information about the search order, see [“The Object Search Order” on page 5-55](#).

Saving Attribute Values

Design Compiler does not automatically save attribute values when you exit `dc_shell`. Use the `write_script` command to generate a `dc_shell` script that re-creates the attribute values.

Note:

The `write_script` command does not support user-defined attributes.

By default, `write_script` prints to the screen. Use the redirection operator (`>`) to redirect the output to a file.

```
dc_shell> write_script > attr.scr
```

Defining Attributes

The `set_attribute` command enables you to create new attributes. Use the `set_attribute` command described in [“Setting Attribute Values” on page 5-51](#).

If you want to change the value of an attribute, remove the attribute and then re-create it to store the desired type.

Removing Attributes

To remove a specific attribute from an object, use the `remove_attribute` command.

You cannot use the `remove_attribute` command to remove inherited attributes. For example, if a `dont_touch` attribute is assigned to a reference, remove the attribute from the reference, not from the cells that inherited the attribute.

For example, to remove the `max_fanout` attribute from port OUT7, enter

```
dc_shell> remove_attribute OUT7 max_fanout
```

You can remove selected attributes by using the `remove_*` commands. Note that some attributes still require the `set_*` command with a `-default` option specified to remove the attribute previously set by the command. See the man page for a specific command to determine whether it has the `-default` option or uses a corresponding `remove` command.

To remove all attributes from the current design, use the `reset_design` command.

```
dc_shell> reset_design  
Resetting current design 'EXAMPLE'.  
1
```

The `reset_design` command removes all design information, including clocks, input and output delays, path groups, operating conditions, timing ranges, and wire load models. The result of using `reset_design` is often equivalent to starting the design process from the beginning.

The Object Search Order

When Design Compiler searches for an object, the search order is command dependent. (Objects include designs, cells, nets, references, and library cells.)

If you do not use a find or get command, Design Compiler uses an implicit find to locate the object. Commands that can set an attribute on more than one type of object use this search order to determine the object to which the attribute applies.

For example, the `set_dont_touch` command operates on cells, nets, references, and library cells. If you define an object, `X`, with the `set_dont_touch` command and two objects (such as the design and a cell) are named `X`, Design Compiler applies the attribute to the first object type found. (In this case, the attribute is set on the design, not on the cell.)

Design Compiler searches until it finds a matching object, or it displays an error message if it does not find a matching object.

In `dcsh` mode, Design Compiler echoes the type of object on which an attribute is set. (If you do not want the echo, set `verbose_messages` to `false`.)

```
dc_shell> set_dont_touch X  
Performing set_dont_touch on design 'X'.  
1
```

You can override the default search order by using the `dcsh` mode `find` command or the `dctcl` mode `get_*` command to specify the object.

For example, assume that the current design contains both a cell and a net named critical. The following command sets the `dont_touch` attribute on the cell because of the default search order:

```
dc_shell> set_dont_touch critical  
Performing set_dont_touch on cell 'critical'.  
1
```

Note:

The status message is not displayed in `dctcl` mode. For predictable results, use the `get_*` command to specify the object in `dctcl` mode.

To place the `dont_touch` attribute on the net instead, use one of the following commands (depending on your shell mode):

```
dc_shell> set_dont_touch find(net, critical)  
Performing set_dont_touch on net 'critical'.  
1
```

```
dc_shell-t> set_dont_touch [get_nets critical]  
1
```

6

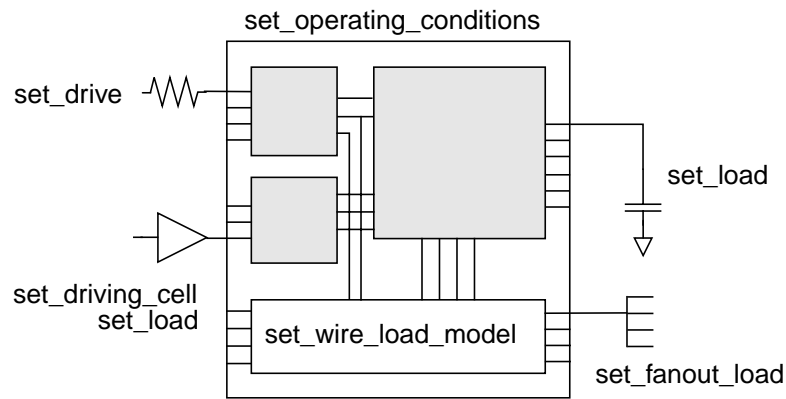
Defining the Design Environment

Before a design can be optimized, you must define the environment in which the design is expected to operate. You define the environment by specifying operating conditions, wire load models, and system interface characteristics.

Operating conditions include temperature, voltage, and process variations. Wire load models estimate the effect of wire length on design performance. System interface characteristics include input drives, input and output loads, and fanout loads. The environment model directly affects design synthesis results.

In Design Compiler, the model is defined by a set of attributes and constraints that you assign to the design, using specific `dc_shell` commands. [Figure 6-1](#) illustrates the commands used to define the design environment.

Figure 6-1 Commands Used to Define the Design Environment



This chapter contains the following sections:

- [Defining the Operating Conditions](#)
- [Defining Wire Load Models](#)
- [Modeling the System Interface](#)

Defining the Operating Conditions

In most technologies, variations in operating temperature, supply voltage, and manufacturing process can strongly affect circuit performance (speed). These factors, called operating conditions, have the following general characteristics:

- Operating temperature variation

Temperature variation is unavoidable in the everyday operation of a design. Effects on performance caused by temperature fluctuations are most often handled as linear scaling effects, but some submicron silicon processes require nonlinear calculations.

- Supply voltage variation

The design's supply voltage can vary from the established ideal value during day-to-day operation. Often a complex calculation (using a shift in threshold voltages) is employed, but a simple linear scaling factor is also used for logic-level performance calculations.

- Process variation

This variation accounts for deviations in the semiconductor fabrication process. Usually process variation is treated as a percentage variation in the performance calculation.

When performing timing analysis, Design Compiler must consider the worst-case and best-case scenarios for the expected variations in the process, temperature, and voltage factors.

Determining Available Operating Condition Options

Most technology libraries have predefined sets of operating conditions. Use the `report_lib` command to list the operating conditions defined in a technology library. The library must be loaded in memory before you can run the `report_lib` command. To see the list of libraries loaded in memory, use the `list_libraries` or the `list_libs` command.

For example, to generate a report for the library `my_lib`, which is stored in `my_lib.db`, enter the following commands:

```
dc_shell> read my_lib.db
dc_shell> report_lib my_lib
```

[Example 6-1](#) shows the resulting operating conditions report.

Example 6-1 Operating Conditions Report

```
*****
Report : library
Library: my_lib
Version: 1999.05
Date   : Mon Jan 4 10:56:49 1999
*****
```

...

Operating Conditions:

Name	Library	Process	Temp	Volt	Interconnect Model
WCCOM	my_lib	1.50	70.00	4.75	worst_case_tree
WCIND	my_lib	1.50	85.00	4.75	worst_case_tree
WCMIL	my_lib	1.50	125.00	4.50	worst_case_tree

...

Specifying Operating Conditions

If the technology library contains operating condition specifications, you can let Design Compiler use them as default conditions.

Alternatively, you can use the `set_operating_conditions` command to specify explicit operating conditions, which supersede the default library conditions.

For example, to set the operating conditions for the current design to worst-case commercial, enter

```
dc_shell> set_operating_conditions WCCOM -lib my_lib
```

Use the `report_design` command to see the operating conditions defined for the current design.

Defining Wire Load Models

Wire load modeling allows you to estimate the effect of wire length and fanout on the resistance, capacitance, and area of nets. Design Compiler uses these physical values to calculate wire delays and circuit speeds.

Semiconductor vendors develop wire load models, based on statistical information specific to the vendors' process. The models include coefficients for area, capacitance, and resistance per unit length, and a fanout-to-length table for estimating net lengths (the number of fanouts determines a nominal length).

Note:

You can also develop custom wire load models. For more information about developing wire load models, see the Library Compiler documentation.

In the absence of back-annotated wire delays, Design Compiler uses the wire load models to estimate net wire lengths and delays. Design Compiler determines which wire load model to use for a design, based on the following factors, listed in order of precedence:

1. Explicit user specification
2. Automatic selection based on design area
3. Default specification in the technology library

If none of this information exists, Design Compiler does not use a wire load model. Without a wire load model, Design Compiler does not have complete information about the behavior of your target technology and cannot compute loading or propagation times for your nets; therefore, your timing information will be optimistic.

In hierarchical designs, Design Compiler must also determine which wire load model to use for nets that cross hierarchical boundaries. The tool determines the wire load model for cross-hierarchy nets based on one of the following factors, listed in order of precedence:

1. Explicit user specification
2. Default specification in the technology library
3. Default mode in Design Compiler

The following sections discuss the selection of wire load models for nets and designs.

Hierarchical Wire Load Models

Design Compiler supports three modes for determining which wire load model to use for nets that cross hierarchical boundaries:

- Top

Design Compiler models nets as if the design has no hierarchy and uses the wire load model specified for the top level of the design hierarchy for all nets in a design and its subdesigns. The tool ignores any wire load models set on subdesigns with the `set_wire_load_model` command.

Use top mode if you plan to flatten the design at a higher level of hierarchy before layout.

- Enclosed

Design Compiler uses the wire load model of the smallest design that fully encloses the net. If the design enclosing the net has no wire load model, the tool traverses the design hierarchy upward until it finds a wire load model. Enclosed mode is more accurate than top mode when cells in the same design are placed in a contiguous region during layout.

Use enclosed mode if the design has similar logical and physical hierarchies.

- Segmented

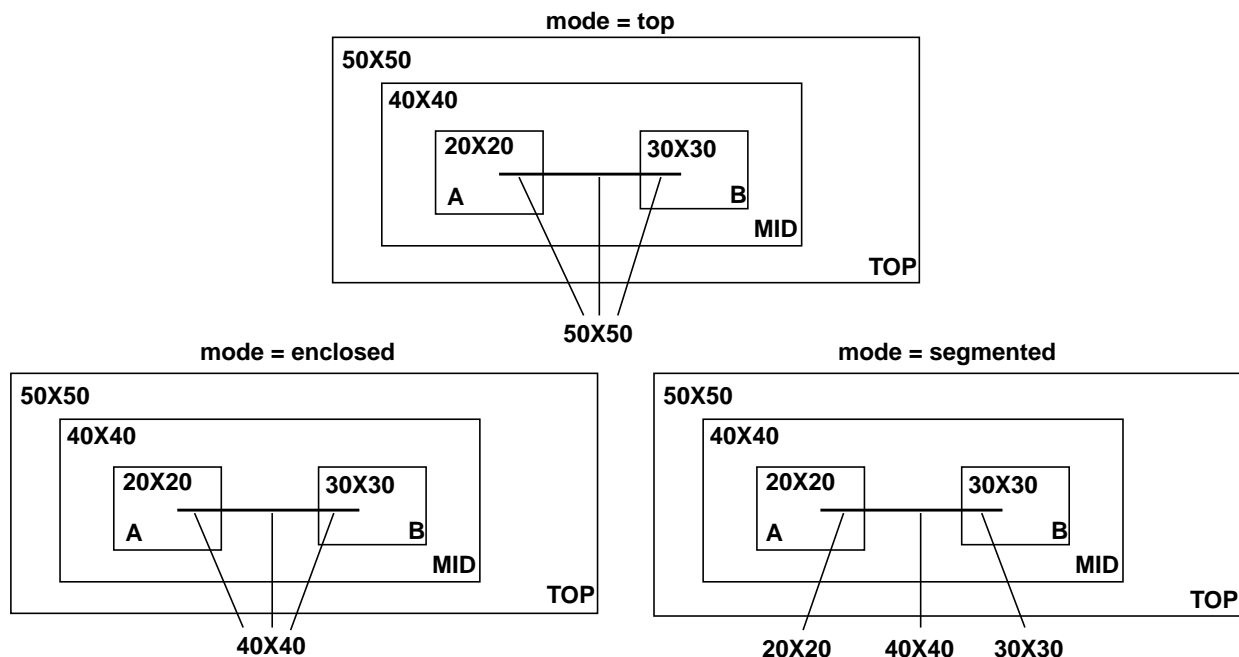
Design Compiler determines the wire load model of each segment of a net by the design encompassing the segment. Nets crossing hierarchical boundaries are divided into segments. For each net segment, Design Compiler uses the wire load model of

the design containing the segment. If the design contains a segment that has no wire load model, the tool traverses the design hierarchy upward until it finds a wire load model.

Use segmented mode if the wire load models in your technology have been characterized with net segments.

Figure 6-2 shows a sample design with a cross-hierarchy net, `cross_net`. The top level of the hierarchy (design TOP) has a wire load model of 50x50. The next level of hierarchy (design MID) has a wire load model of 40x40. The leaf-level designs, A and B, have wire load models of 20x20 and 30x30, respectively.

Figure 6-2 Comparison of Wire Load Mode



In top mode, Design Compiler estimates the wire length of net `cross_net`, using the 50x50 wire load model. Design Compiler ignores the wire load models on designs MID, A, and B.

In enclosed mode, Design Compiler estimates the wire length of net `cross_net`, using the 40x40 wire load model (the net `cross_net` is completely enclosed by design MID).

In segmented mode, Design Compiler uses the 20x20 wire load model for the net segment enclosed in design A, the 30x30 wire load model for the net segment enclosed in design B, and the 40x40 wire load model for the segment enclosed in design MID.

Determining Available Wire Load Models

Most technology libraries have predefined wire load models. Use the `report_lib` command to list the wire load models defined in a technology library. The library must be loaded in memory before you run the `report_lib` command. To see a list of libraries loaded in memory, use the `list_libs` command.

The wire load report contains the following sections:

- Wire Loading Model section

This section lists the available wire load models.

- Wire Loading Model Mode section

This section identifies the default wire load mode. If a library default does not exist, Design Compiler uses top mode.

- Wire Loading Model Selection Group section

The presence of this section indicates that the library supports automatic area-based wire load model selection.

To generate a wire load report for the `my_lib` library, enter

```
dc_shell> read my_lib.db
```

```
dc_shell> report_lib my_lib
```

Example 6-2 shows the resulting wire load models report. The library `my_lib` contains three wire load models: 05x05, 10x10, and 20x20. The library does not specify a default wire load mode (so Design Compiler uses top as the default wire load mode), and it supports automatic area-based wire load model selection.

Example 6-2 Wire Load Models Report

```
*****
Report : library
Library: my_lib
Version: 1999.05
Date   : Mon Jan 4 10:56:49 1999
*****
...
Wire Loading Model:

Name       : 05x05
Location   : my_lib
Resistance : 0
Capacitance : 1
Area       : 0
Slope      : 0.186
Fanout     Length  Points Average Cap Std Deviation
-----
1          0.39

Name       : 10x10
Location   : my_lib
Resistance : 0
Capacitance : 1
Area       : 0
Slope      : 0.311
Fanout     Length  Points Average Cap Std Deviation
-----
1          0.53
```

Example 6-2 Wire Load Models Report (Continued)

```
Name           : 20x20
Location        : my_lib
Resistance      : 0
Capacitance     : 1
Area            : 0
Slope           : 0.547
Fanout   Length   Points Average Cap Std Deviation
-----
```

```
1           0.86
```

Wire Loading Model Selection Group:

```
Name           : my_lib
```

Selection		Wire load name
min area	max area	
0.00	1000.00	05x05
1000.00	2000.00	10x10
2000.00	3000.00	20x20

...

Specifying Wire Load Models and Modes

The technology library can define a default wire load model that is used for all designs implemented in that technology. The `default_wire_load` library attribute identifies the default wire load model for a technology library.

Some libraries support automatic area-based wire load selection. Design Compiler uses the library function `wire_load_selection` to choose a wire load model based on the total cell area.

For large designs with many levels of hierarchy, automatic wire load selection can increase runtime. To manage runtime, set the wire load manually.

You can turn off automatic selection of the wire load model by setting the `auto_wire_load_selection` variable to false. For example, enter one of the following commands (depending on your shell mode):

```
dc_shell> auto_wire_load_selection = false
```

```
dc_shell-t> set auto_wire_load_selection false
```

The technology library can also define a default wire load mode. The `default_wire_load_mode` library attribute identifies the default mode. If the current library does not define a default mode, Design Compiler looks for the attribute in the libraries specified in the `link_library` variable. (To see the link library, use the `list` command.) In the absence of a library default (and an explicit specification), Design Compiler uses that top mode.

To change the wire load model or mode specified in a technology library, use the `set_wire_load_model` and `set_wire_load_mode` commands. The wire load model and mode you define override all defaults. Explicitly selecting a wire load model also disables area-based wire load model selection for that design.

For example, to select the 10x10 wire load model, enter

```
dc_shell> set_wire_load_model "10x10"
```

To select the 10x10 wire load model and specify enclosed mode, enter

```
dc_shell> set_wire_load_mode enclosed
```

The wire load model you choose for a design depends on how that design is implemented in the chip. Consult your semiconductor vendor to determine the best wire load model for your design.

Use the `report_design` or `report_timing` commands to see the wire load model and mode defined for the current design.

To remove the wire load model, use the `remove_wire_load_model` command with no model name.

Modeling the System Interface

Design Compiler supports the following ways to model the design's interaction with the external system:

- Defining drive characteristics for input ports
- Defining loads on input and output ports
- Defining fanout loads on output ports

The following sections discuss these tasks.

Defining Drive Characteristics for Input Ports

Design Compiler uses drive strength information to buffer nets appropriately in the case of a weak driver.

Note:

Drive strength is the reciprocal of the output driver resistance, and the transition time delay at an input port is the product of the drive resistance and the capacitance load of the input port.

By default, Design Compiler assumes zero drive resistance on input ports, meaning infinite drive strength. There are three commands for overriding this unrealistic assumption:

- `set_driving_cell`

- `set_drive`
- `set_input_transition`

Both the `set_driving_cell` and `set_input_transition` commands affect the port transition delay, but they do not place design rule requirements, such as `max_fanout` and `max_transition`, on input ports. However, the `set_driving_cell` command does place design rules on input ports if the driving cell has DRCs.

Note:

For heavily loaded driving ports, such as clock lines, keep the drive strength setting at 0 so that Design Compiler does not buffer the net. Each semiconductor vendor has a different way of distributing these signals within the silicon.

Both the `set_drive` and the `set_driving_cell` commands affect the port transition delay. The `set_driving_cell` command can place design rule requirements, such as `max_fanout` or `max_transition`, on input ports if the specified cell has input ports.

The most recently used command takes precedence. For example, setting a drive resistance on a port with the `set_drive` command overrides previously run `set_driving_cell` commands.

The `set_driving_cell` Command

Use the `set_driving_cell` command to specify drive characteristics on ports that are driven by cells in the technology library. This command is compatible with all the delay models, including the nonlinear delay model and piecewise linear delay

model. The `set_driving_cell` command associates a library pin with an input port so that delay calculators can accurately model the drive capability of an external driver.

To return to the default of zero drive resistance, use the `set_driving_cell -none` command.

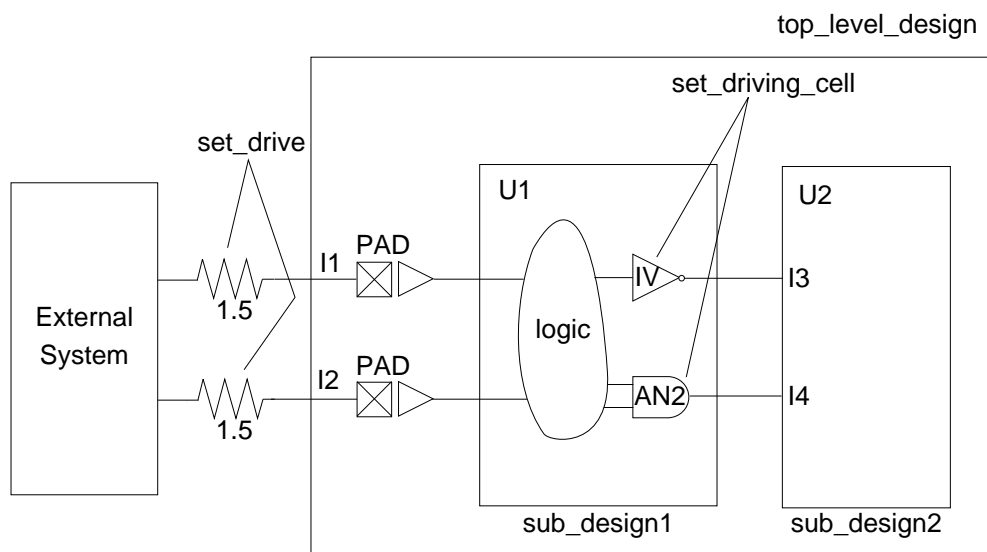
The `set_drive` and `set_input_transition` Commands

Use the `set_drive` or `set_input_transition` command to set the drive resistance on the top-level ports of the design when the input port drive capability cannot be characterized with a cell in the technology library.

You can use `set_drive` and the `drive_of` commands together to represent the drive resistance of a cell. However, these commands are not as accurate for nonlinear delay models as the `set_driving_cell` command is.

[Figure 6-3](#) shows a hierarchical design. The top-level design has two subdesigns, U1 and U2. Ports I1 and I2 of the top-level design are driven by the external system and have a drive resistance of 1.5.

Figure 6-3 Drive Characteristics



To set the drive characteristics for this example, follow these steps:

1. Because ports I1 and I2 are not driven by library cells, use the `set_drive` command to define the drive resistance. Enter

```
dc_shell> current_design top_level_design  
dc_shell> set_drive 1.5 {I1 I2}
```
2. To describe the drive capability for the ports on design `sub_design2`, change the current design to `sub_design2`. Enter

```
dc_shell> current_design sub_design2
```
3. An IV cell drives port I3. Use the `set_driving_cell` command to define the drive resistance. Because IV has only one output and one input, define the drive capability as follows. Enter

```
dc_shell> set_driving_cell -cell IV {I3}
```

4. An AN2 cell drives port I4. Because the different arcs of this cell have different transition times, select the worst-case arc to define the drive. For checking setup violations, the worst-case arc is the slowest arc. For checking hold violations, the worst-case arc is the fastest arc.

For this example, assume that you want to check for setup violations. The slowest arc on the AN2 cell is the B-to-Z arc, so define the drive as follows. Enter

```
dc_shell> set_driving_cell -cell AN2 -pin Z \  
          -from_pin B {I4}
```

Defining Loads on Input and Output Ports

By default, Design Compiler assumes zero capacitive load on input and output ports. Use the `set_load` command to set a capacitive load value on input and output ports of the design. This information helps Design Compiler select the appropriate cell drive strength of an output pad and helps model the transition delay on input pads.

For example, to set a load of 30 on output pin out1, enter

```
dc_shell> set_load 30 {out1}
```

Make the units for the load value consistent with the target technology library. For example, if the library represents the load value in picofarads, the value you set with the `set_load` command must be in picofarads. Use the `report_lib` command to list the library units.

[Example 6-3](#) shows the library units for the library `my_lib`.

Example 6-3 *Library Units Report*

```
*****
Report : library
Library: my_lib
Version: 1999.05
Date   : Mon Jan 4 10:56:49 1999
*****

Library Type           : Technology
Tool Created           : 1999.05
Date Created           : February 7, 1992
Library Version        : 1.800000
Time Unit               : 1ns
Capacitive Load Unit   : 0.100000ff
Pulling Resistance Unit : 1kilo-ohm
Voltage Unit            : 1V
Current Unit            : 1uA
...
```

Defining Fanout Loads on Output Ports

You can model the external fanout effects by specifying the expected fanout load values on output ports with the `set_fanout_load` command.

For example, enter

```
dc_shell> set_fanout_load 4 {out1}
```

Design Compiler tries to ensure that the sum of the fanout load on the output port plus the fanout load of cells connected to the output port driver is less than the maximum fanout limit of the library, library cell, and design. (For more information about maximum fanout limits, see [“Setting Design Rule Constraints” on page 7-3.](#))

Fanout load is not the same as load. Fanout load is a unitless value that represents a numerical contribution to the total fanout. Load is a capacitance value. Design Compiler uses fanout load primarily to measure the fanout presented by each input pin. An input pin normally has a fanout load of 1, but it can have a higher value.

7

Defining Design Constraints

In addition to specifying the design environment, you must set design constraints before compiling the design. There are two categories of design constraints:

- Design rule constraints
- Design optimization constraints

Design rule constraints are supplied in the technology library you specify. They are referred to as the *implicit* design rules. These rules are established by the library vendor, and, for the proper functioning of the fabricated circuit, they must not be violated. You can, however, specify stricter design rules if appropriate. The rules you specify are referred to as the *explicit* design rules.

Design optimization constraints define timing and area optimization goals for Design Compiler. These constraints are user-specified. Design Compiler optimizes the synthesis of the design, in

accordance with these constraints, but not at the expense of the design rule constraints. That is, Design Compiler attempts never to violate the higher-priority design rules.

Note:

In this chapter, setting explicit design rules and optimization constraints is discussed without reference to the particular compile strategy you choose. But the compile strategy you choose does influence your constraint settings.

This chapter contains the following sections:

- [Setting Design Rule Constraints](#)
- [Setting Optimization Constraints](#)
- [Verifying the Precompiled Design](#)

The task of setting timing constraints can be complicated (especially setting the timing exceptions) and includes the following tasks:

- [Defining a Clock](#)
- [Specifying I/O Timing Requirements](#)
- [Specifying Combinational Path Delay Requirements](#)
- [Specifying Timing Exceptions](#)

Setting Design Rule Constraints

This section discusses the most commonly specified design rule constraints:

- Transition time
- Fanout load
- Capacitance

Design Compiler also supports cell degradation and connection class constraints. For information about these constraints, see the *Design Compiler Reference Manual: Constraints and Timing*.

Design Compiler uses attributes assigned to the design's objects to represent design rule constraints. [Table 7-1](#) provides the attribute name that corresponds to each design rule constraint.

Table 7-1 Design Rule Attributes

Design rule constraint	Attribute name
Transition time	max_transition
Fanout load	max_fanout
Capacitance	max_capacitance min_capacitance
Cell degradation	cell_degradation
Connection class	connection_class

Design rule constraints are attributes specified in the technology library and, optionally, specified by you explicitly.

If a technology library defines these attributes, Design Compiler implicitly applies them to any design using that library when it compiles the design or creates a constraint report. You cannot remove the design rule attributes defined in the technology library, because they are requirements for the technology, but you can make them more restrictive to suit your design.

If both implicit and explicit design rule constraints apply to a design or a net, the more restrictive value takes precedence.

Setting Transition Time Constraints

The transition time of a net is the time required for its driving pin to change logic values. This transition time is based on the technology library data. For the nonlinear delay model (NLDM), output transition time is a function of input transition and output load.

Design Compiler and Library Compiler model transition time restrictions by associating a `max_transition` attribute with each output pin on a cell. During optimization, Design Compiler attempts to make the transition time of each net less than the value of the `max_transition` attribute.

To change the maximum transition time restriction specified in a technology library, use the `set_max_transition` command. This command sets a maximum transition time for the nets attached to the identified ports or to all the nets in a design by setting the `max_transition` attribute on the named objects.

For example, to set a maximum transition time of 3.2 on all nets in the design adder, enter one of the following commands (depending on your shell mode):

```
dc_shell> set_max_transition 3.2 find(design,adder)  
dc_shell-t> set_max_transition 3.2 [get_designs adder]
```

To undo a `set_max_transition` command, use the `remove_attribute` command. For example, enter one of the following commands (depending on your shell mode):

```
dc_shell>remove_attribute find(design,adder) \  
          max_transition  
dc_shell-t>remove_attribute [get_designs adder] \  
          max_transition
```

Setting Fanout Load Constraints

The maximum fanout load for a net is the maximum number of loads the net can drive.

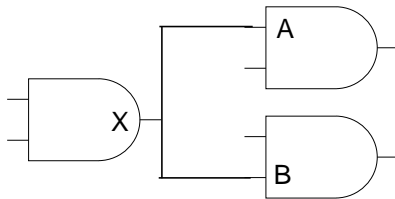
Design Compiler and Library Compiler model fanout restrictions by associating a `fanout_load` attribute with each input pin and a `max_fanout` attribute with each output (driving) pin on a cell.

The fanout load value does not represent capacitance; it represents the weighted numerical contribution to the total fanout load. The fanout load imposed by an input pin is not necessarily 1.0. Library developers can assign higher fanout load values to model internal cell fanout effects.

Design Compiler calculates the fanout of a driving pin by adding the `fanout_load` values of all inputs driven by that pin. To determine whether the pin meets the maximum fanout load restriction, Design Compiler compares the calculated fanout load value with the pin's `max_fanout` value.

Figure 7-1 shows a small circuit in which pin X drives two loads, pin A and pin B. If pin A has a `fanout_load` value of 1.0 and pin B has a `fanout_load` value of 2.0, the total fanout load of pin X is 3.0. If pin X has a maximum fanout greater than 3.0, say 16.0, the pin meets the fanout constraints.

Figure 7-1 Fanout Constraint Example



During optimization, Design Compiler attempts to meet the fanout load restrictions for each driving pin. If a pin violates its fanout load restriction, Design Compiler tries to correct the problem (for example, by changing the drive strength of the component).

The technology library might specify default fanout constraints on the entire library or fanout constraints for specific pins in the library description of an individual cell.

To determine whether your technology library is modeled for fanout calculations, you can search for the `fanout_load` attribute on the cell input pins by entering one of the following commands (depending on your shell mode):

```
dc_shell> get_attribute find(pin, my_lib/*/*) fanout_load
```

```
dc_shell-t> get_attribute [get_pins my_lib/*/*] fanout_load
```

To set a more conservative fanout restriction than that specified in the technology library, use the `set_max_fanout` command on the design or on an input port. (Use the `set_fanout_load` command to set the expected fanout load value for output ports.)

The `set_max_fanout` command sets the maximum fanout load for the specified input ports or for all the nets in a design by setting the `max_fanout` attribute on the specified objects. For example, to set a `max_fanout` requirement of 16 on all nets in the design adder, enter one of the following commands (depending on your shell mode):

```
dc_shell> set_max_fanout 16 find(design, adder)
```

```
dc_shell-t> set_max_fanout 16 [get_designs adder]
```

If you use the `set_max_fanout` command and a library `max_fanout` attribute exists, Design Compiler tries to meet the smaller (more restrictive) fanout limit.

To undo a `set_max_fanout` command, use the `remove_attribute` command. For example, enter one of the following commands (depending on your shell mode):

```
dc_shell> remove_attribute find(design,adder) max_fanout
```

```
dc_shell-t> remove_attribute [get_designs adder] max_fanout
```

Setting Capacitance Constraints

The transition time constraints do not provide a direct way to control the actual capacitance of nets. If you need to control capacitance directly, use the `set_max_capacitance` command to set the maximum capacitance constraint. This constraint is completely independent, so you can use it in addition to the transition time constraints.

Design Compiler and Library Compiler model capacitance restrictions by associating the `max_capacitance` attribute with the output ports or pins of a cell. Design Compiler calculates the capacitance on the output net by adding the wire capacitance of the net to the capacitance of the pins attached to the net. To determine whether the net meets the capacitance constraint, Design Compiler compares the calculated capacitance value with the output pin's `max_capacitance` value.

For example, to set a maximum capacitance of 3 for all nets in the design adder, enter one of the following commands (depending on your shell mode):

```
dc_shell> set_max_capacitance 3 find(design,adder)
```

```
dc_shell-t> set_max_capacitance 3 [get_designs adder]
```

To undo a `set_max_capacitance` command, use the `remove_attribute` command. For example, enter one of the following commands (depending on your shell mode):

```
dc_shell> remove_attribute find(design,adder) \  
          max_capacitance
```

```
dc_shell-t> remove_attribute [get_designs adder] \  
          max_capacitance
```

You can also use the `set_min_capacitance` command to define the minimum capacitance for input ports or pins. Design Compiler attempts to ensure that the load seen at the input port does not fall below the specified capacitance value, but it does not specifically optimize for this constraint.

Setting Optimization Constraints

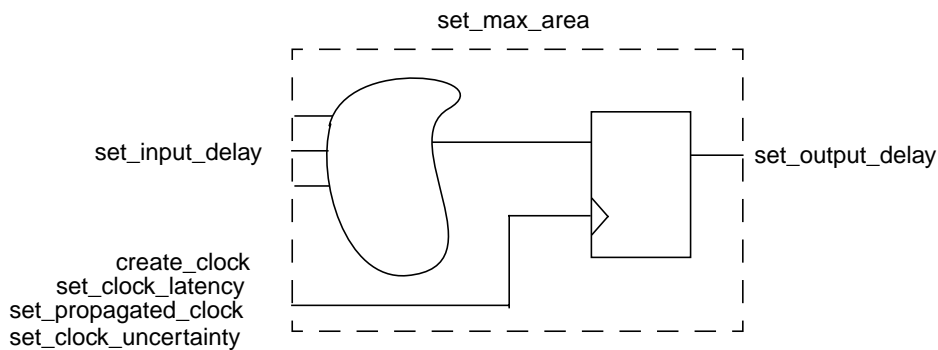
This section discusses the most commonly specified optimization constraints:

- Timing constraints
- Area constraints

Design Compiler also supports power constraints. For information about power constraints, see the *Power Compiler Reference Manual*.

Figure 7-2 illustrates some of the common commands used to define the optimization constraints.

Figure 7-2 Commands Used to Define the Optimization Constraints for Sequential Blocks



Setting Timing Constraints

Timing constraints specify the required performance of the design. To set the timing constraints,

1. Define the clocks.

2. Specify the I/O timing requirements relative to the clocks.
3. Specify the combinational path delay requirements.
4. Specify the timing exceptions.

[Table 7-2](#) lists the most commonly used commands for these steps.

Table 7-2 Commands to Set Timing Constraints

Command	Description
<code>create_clock</code>	Defines the period and waveform for the clock.
<code>set_clock_latency</code> <code>set_propagated_clock</code> <code>set_clock_uncertainty</code>	Defines the clock delay.
<code>set_input_delay</code>	Defines the timing requirements for input ports relative to the clock period.
<code>set_output_delay</code>	Defines the timing requirements for output ports relative to the clock period.
<code>set_max_delay</code>	Defines maximum delay for combinational paths. (This is a timing exception command.)
<code>set_min_delay</code>	Defines minimum delay for combinational paths. (This is a timing exception command.)
<code>set_false_path</code>	Specifies false paths. (This is a timing exception command.)
<code>set_multicycle_path</code>	Specifies multicycle paths. (This is a timing exception command.)

The following sections describe these steps in more detail.

Defining a Clock

For synchronous designs, the clock period is the most important constraint because it constrains all register-to-register paths in the design.

Defining the Period and Waveform for the Clock. Use the `create_clock` command to define the period (`-period` option) and waveform (`-waveform` option) for the clock. If you do not specify the clock waveform, Design Compiler uses a 50 percent duty cycle.

Use the `create_clock` command on a pin or a port. For example, to specify a 25-megahertz clock on port `clk` with a 50 percent duty cycle, enter

```
dc_shell> create_clock clk -period 40
```

When your design contains multiple clocks, pay close attention to the common base period of the clocks. The common base period is the least common multiple of all the clock periods. For example, if you have clock periods of 10, 15, and 20, the common base period is 60.

Define your clocks so that the common base period is a small integer multiple of each of the clock periods. The common base period requirement is qualitative; no hard limit exists. If the base period is more than 10 times larger than the smallest period, however, long runtimes and greater memory requirements can result.

As an extreme case, if you have a register-to-register path where one register has a period of 10 and the other has a period of 10.1, the common base period is 1010.0. The timing analyzer calculates the setup requirement for this path by expanding both clocks to the common base period and determining the tightest single-cycle

relationship for setup. Internally, for extreme cases such as this, the timing analyzer only approximates the setup requirement because the paths are not really synchronous.

You can work around this problem by specifying a clock period without a decimal point and adjusting the clock period by inserting clock uncertainty.

```
dc_shell> create_clock -period 10 clk1
dc_shell> create_clock -period 10 clk2
dc_shell> set_clock_uncertainty -setup 0.1 clk2
```

Use the `report_clock` command to show information about all clock sources in your design.

Use the `remove_clock` command to remove a clock definition.

Creating a Virtual Clock. In some cases, a system clock might not exist in a block. You can use the `create_clock -name` command to create a virtual clock for modeling clock signals present in the system but not in the block. By creating a virtual clock, you can represent delays that are relative to clocks outside the block.

```
dc_shell> create_clock -period 30 -waveform {10 25} \
           -name sys_clk
```

Specifying Clock Network Delay. By default, Design Compiler assumes that clock networks have no delay (ideal clocks). Use the `set_clock_latency` and `set_clock_uncertainty` commands to specify timing information about the clock network delay. You can use these commands to specify either estimated or actual delay information.

Use the `set_propagated_clock` command to specify that you want the clock latency to propagate through the clock network. For example,

```
dc_shell> set_propagated_clock clk
```

Use the `-setup` or `-hold` options of the `set_clock_uncertainty` command to add some margin of error into the system to account for variances in the clock network resulting from layout. For example, on the 20-megahertz clock mentioned previously, to add a 0.2 margin on each side of the clock edge, enter

```
dc_shell> set_clock_uncertainty -setup 0.2 clk  
dc_shell> set_clock_uncertainty -hold 0.2 clk
```

Use the `-skew` option of the `report_clock` command to show clock network skew information. Design Compiler uses the clock information when determining whether a path meets setup and hold requirements.

Specifying I/O Timing Requirements

If you do not assign timing requirements to an input port, Design Compiler responds as if the signal arrives at the input port at time 0. In most cases, input signals arrive at staggered times. Use the `set_input_delay` command to define the arrival times for input ports. You define the input delay constraint relative to the system clock and to the other inputs.

If you do not assign timing requirements to an output port, Design Compiler does not constrain any paths which end at an output port. Use the `set_output_delay` command to define the required output arrival time. You define the output delay constraint relative to the system clock.

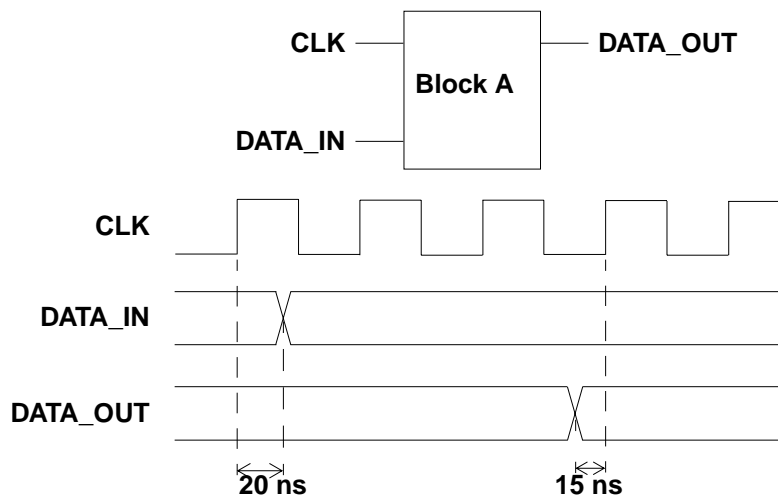
If an input or output port has multiple timing requirements (because of multiple paths), use the `-add_delay` option to specify the additional timing requirements.

Use the `report_port` command to list input or output delays associated with ports.

Use the `remove_input_delay` command to remove input delay constraints. Use the `remove_output_delay` command to remove output delay constraints.

Figure 7-3 shows the timing relationship between the delay and the active clock edge (the rising edge in this example).

Figure 7-3 Relationship Between Delay and Active Clock Edge



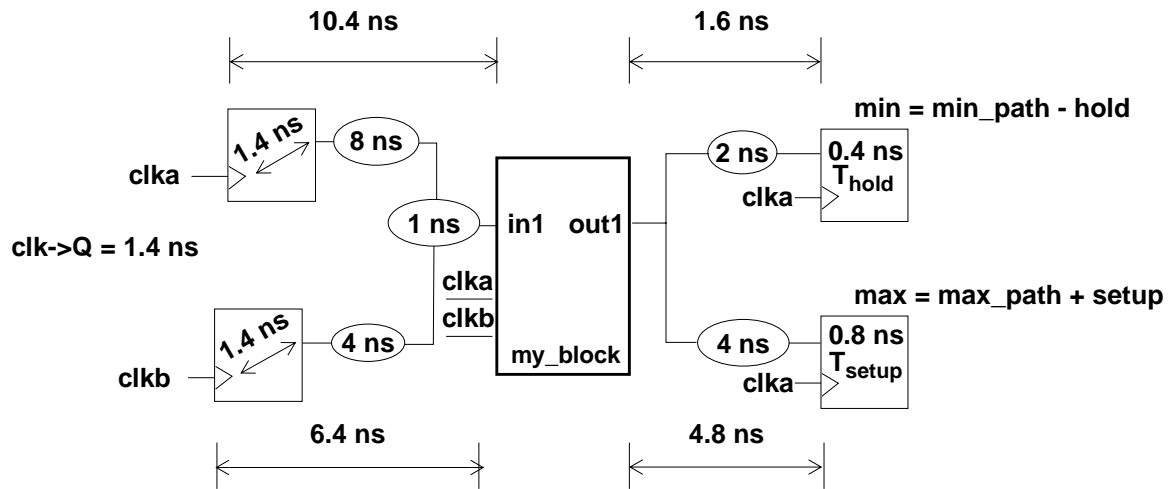
In the figure, block A has an input DATA_IN and an output DATA_OUT. From the waveform diagram, DATA_IN is stable 20 ns after the clock edge, and DATA_OUT needs to be available 15 ns before the clock edge.

After you set the clock constraint by using the `create_clock` command, use the `set_input_delay` and `set_output_delay` commands to specify these additional requirements. For example, enter

```
dc_shell> set_input_delay 20 -clock CLK DATA_IN
dc_shell> set_output_delay 15 -clock CLK DATA_OUT
```

Figure 7-4 illustrates the timing requirements for the constrained design block `my_block`. Example 7-1 shows the script used to specify these timing requirements.

Figure 7-4 Timing Requirements for `my_block`



Example 7-1 Timing Constraints for `my_block`

```
create_clock -period 20 -waveform {5 15} clka
create_clock -period 30 -waveform {10 25} clkb
set_input_delay 10.4 -clock clka in1
set_input_delay 6.4 -clock clkb -add_delay in1
set_output_delay 1.6 -clock clka -min out1
set_output_delay 4.8 -clock clka -max out1
```

Specifying Combinational Path Delay Requirements

For purely combinational delays that are not bounded by a clock period, use the `set_max_delay` and `set_min_delay` commands to define the maximum and minimum delays for the specified paths.

A common way to produce this type of asynchronous logic in HDL code is to use asynchronous sets or resets on latches and flip-flops. Because the reset signal crosses several blocks, constrain this signal at the top level.

For example, to specify a maximum delay of 5 on the RESET signal, enter

```
dc_shell> set_max_delay 5 -from RESET
```

To specify a minimum delay of 10 on the path from IN1 to OUT1, enter

```
dc_shell> set_min_delay 10 -from IN1 -to OUT1
```

Use the `report_timing_requirements` command to list the minimum delay and maximum delay requirements for your design.

Specifying Timing Exceptions

Timing exceptions define timing relationships that override the default single-cycle timing relationship for one or more timing paths. Use timing exceptions to constrain or disable asynchronous paths or paths that do not follow the default single-cycle behavior.

Note:

Specifying numerous timing exceptions can increase the compile runtime. Nevertheless, some designs can require many timing exceptions.

Design Compiler recognizes only timing exceptions that have valid reference points.

- The valid startpoints in a design are the primary input ports and the clock pins of sequential cells.
- The valid endpoints are the primary output ports of a design and the data pins of sequential cells.

Design Compiler does not generate a warning message if you specify invalid reference points. You must use the `-ignored` option of the `report_timing_requirements` command to find timing exceptions ignored by Design Compiler.

You can specify the following conditions by using timing exception commands:

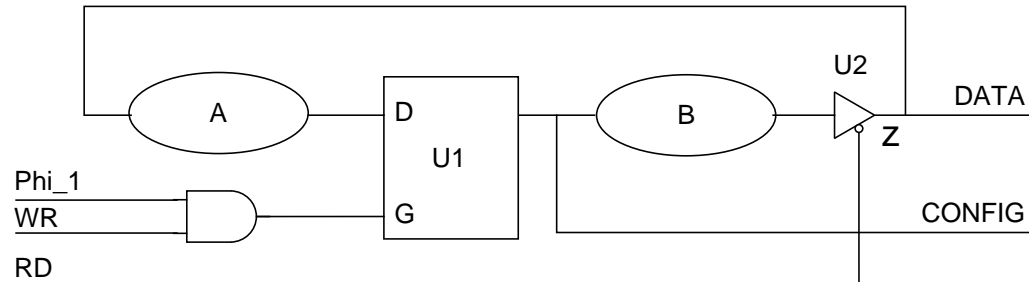
- False paths (`set_false_path`)
- Minimum delay requirements (`set_min_delay`)
- Maximum delay requirements (`set_max_delay`)
- Multicycle paths (`set_multicycle_path`)

Use the `report_timing_requirements` command to list the timing exceptions in your design.

Specifying False Paths. Design Compiler does not report false paths in the timing report or consider them during timing optimization. Use the `set_false_path` command to specify a false path. Use this command to ignore paths that are not timing critical, that can mask other paths that must be considered during optimization, or that never occur in normal operation.

For example, [Figure 7-5](#) shows a configuration register that can be written and read from a bidirectional bus (DATA) in a chip.

Figure 7-5 Configuration Register



The circuit has these timing paths:

1. DATA to U1/D
2. RD to DATA
3. U1/G to CONFIG (with possible time borrowing at U1/D)
4. U1/G to DATA (with possible time borrowing at U1/D)
5. U1/G to U1/D (through DATA, with possible time borrowing)

The first four paths are valid paths. The fifth path (U1/G to U1/D) is a functional false path because normal operation never requires simultaneous writing and reading of the configuration register. In this design, you can disable the false path by using this command:

```
dc_shell> set_false_path -from U1/G -to U1/D
```

To undo a `set_false_path` command, use the `reset_path` command with similar options. For example, enter

```
dc_shell> set_false_path -setup -from IN2 -to FF12/D
dc_shell> reset_path -setup -from IN2 -to FF12/D
```

Creating a false path differs from disabling a timing arc. Disabling a timing arc represents a break in the path. The disabled timing arc permanently disables timing through all affected paths. Specifying a path as false does not break the path; it just prevents the path from being considered for timing or optimization.

Specifying Minimum and Maximum Delay Requirements. You can use the `set_min_delay` and `set_max_delay` commands, described earlier in this chapter, to specify path delay requirements that are more conservative than those derived by Design Compiler based on the clock timing.

To undo a `set_min_delay` or `set_max_delay` command, use the `reset_path` command with similar options.

Register-to-Register Paths. Design Compiler uses the following equations to derive constraints for minimum and maximum path delays on register-to-register paths:

$$\begin{aligned}\text{min_delay} &= (T_{\text{capture}} - T_{\text{launch}}) + \text{hold} \\ \text{max_delay} &= (T_{\text{capture}} - T_{\text{launch}}) - \text{setup}\end{aligned}$$

You can override the derived path delay ($T_{\text{capture}} - T_{\text{launch}}$) by using the `set_min_delay` and `set_max_delay` commands.

For example, assume that you have a path launched from a register at time 20 that arrives at a register where the next active edge of the clock occurs at time 35.

```
dc_shell> create_clock -period 40 waveform {0 20} clk1
dc_shell> create_clock -period 40 -waveform {15 35} clk2
```

Design Compiler automatically derives a maximum path delay constraint of $(35 - 20) - (\text{library setup time of register at endpoint})$. To specify a maximum path delay of 10, enter

```
dc_shell> set_max_delay 10 -from reg1 -to reg2
```

Design Compiler calculates the maximum path delay constraint as $10 - (\text{library setup time of register at endpoint})$, which overrides the original derived maximum path delay constraint.

Register-to-Port Paths. Design Compiler uses the following equations to derive constraints for minimum and maximum path delays on register-to-port paths:

```
min_delay = period - output_delay  
max_delay = period - output_delay
```

If you use the `set_min_delay` or `set_max_delay` commands, the value specified in these commands replaces the period value in the constraint calculation. For example, assume you have a design with a clock period of 20. Output OUTPORTA has an output delay of 5.

```
dc_shell> create_clock -period 20 CLK  
dc_shell> set_output_delay 5 -clock CLK OUTPORTA
```

Design Compiler automatically derives a maximum path delay constraint of 15 ($20 - 5$). To specify that you want a maximum path delay of 10, enter

```
dc_shell> set_max_delay 10 -to OUTPORTA
```

Design Compiler calculates the maximum path delay constraint as 5 ($10 - 5$), which overrides the original derived maximum path delay constraint.

Asynchronous Paths. You can also use the `set_max_delay` and `set_min_delay` commands to constrain asynchronous paths across different frequency domains. [Table 7-3](#) shows `dcsh` and `dctcl` examples for constraining asynchronous paths.

Table 7-3 Examples for Constraining Asynchronous Paths

dcsh Example		dctcl Example	
<code>dc_shell> set_max_delay 17.1 \</code>	<code>-from</code>	<code>dc_shell-t> set_max_delay 17.1 \</code>	<code>-from</code>
<code>find(clock, clk1) \</code>		<code>[get_clocks clk1] \</code>	
<code>-to find(clock, clk2)</code>		<code>-to [get_clocks clk2]</code>	
<code>dc_shell> set_max_delay 23.5 \</code>	<code>-from</code>	<code>dc_shell-t> set_max_delay 23.5 \</code>	<code>-from</code>
<code>find(clock, clk2) \</code>		<code>[get_clocks clk2] \</code>	
<code>-to find(clock, clk3)</code>		<code>-to [get_clocks clk3]</code>	
<code>dc_shell> set_max_delay 31.6 \</code>	<code>-from</code>	<code>dc_shell-t> set_max_delay 31.6 \</code>	<code>-from</code>
<code>find(clock, clk3) \</code>		<code>[get_clocks clk3] \</code>	
<code>-to find(clock, clk1)</code>		<code>-to [get_clocks clk1]</code>	

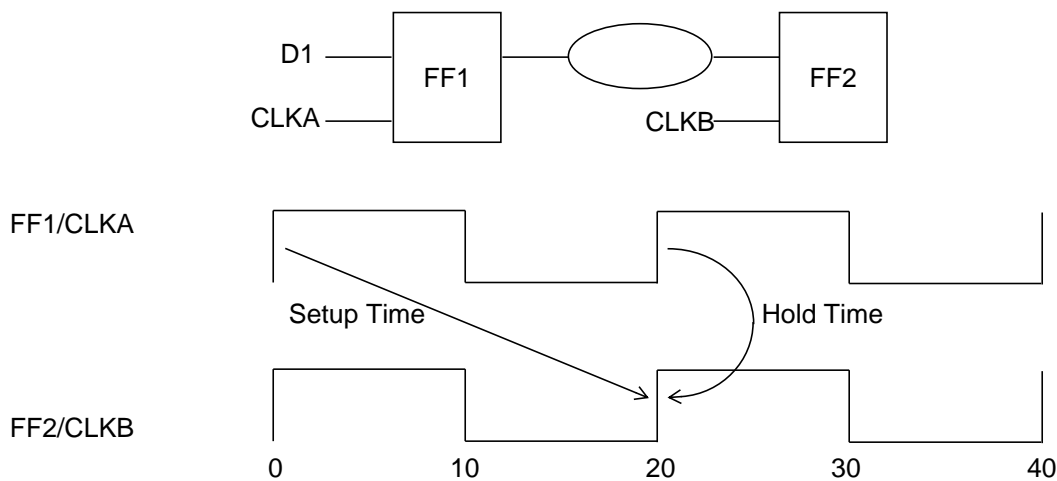
Setting Multicycle Paths. The multicycle path condition is appropriate when the path in question is longer than a single cycle or when data is not expected within a single cycle. Use the `set_multicycle_path` command to specify the number of clock cycles Design Compiler should use to determine when data is required at a particular endpoint.

You can specify this cycle multiplier for setup or hold checks. If you do not specify the `-setup` or `-hold` option with the `set_multicycle_path` command, Design Compiler applies the multiplier value only to setup checks.

By default, setup is checked at the next active edge of the clock at the endpoint after the data is launched from the startpoint (default multiplier of 1). Hold data is launched one clock cycle after the setup data but checked at the edge used for setup (default multiplier of zero).

[Figure 7-6](#) shows the timing relationship of setup and hold times.

Figure 7-6 Setup and Hold Timing



The timing path starts at the clock pin of FF1 (rising edge of CLKA) and ends at the data pin of FF2. Assuming that the flip-flops are rising-edge-triggered, the setup data is launched at time 0 and checked 20 time units later at the next active edge of CLKB at FF2. Hold data is launched one (CLKA) clock cycle (time 20) and checked at the same edge used for setup checking (time 20).

The `-setup` option of the `set_multicycle_path` command moves the edge used for setup checking to before or after the default edge. For the example shown in [Figure 7-6](#),

- A setup multiplier of zero means that Design Compiler uses the edge at time zero for checking

- A setup multiplier of 2 means that Design Compiler uses the edge at time 40 for checking

The `-hold` option of the `set_multicycle_path` command launches the hold data at the edge before or after the default edge, but Design Compiler still checks the hold data at the edge used for checking setup. As shown in [Figure 7-6](#) (assuming a default setup multiplier),

- A hold multiplier of 1 means that the hold data is launched from CLKA at time 40 and checked at CLKB at time 20
- A hold multiplier of -1 means that the hold data is launched from CLKA at time 0 and checked at CLKB at time 20

To undo a `set_multicycle_path` command, use the `reset_path` command with similar options.

Using Multiple Timing Exception Commands. A specific timing exception command refers to a single timing path. A general timing exception command refers to more than one timing path. If you execute more than one instance of a given timing exception command, the more specific commands override the more general ones.

The following rules define the order of precedence for a given timing exception command:

- The highest precedence occurs when you define a timing exception from one pin to another pin.
- A command using only the `-from` option has a higher priority than a command using only the `-to` option.

- For clocks used in timing exception commands, if both `-from` and `-to` are defined, they override commands that share the same path defined by either the `-from` or the `-to` option.

This list details the order of precedence (highest at the top) defined by these precedence rules:

1. *command* -from *pin* -to *pin*
2. *command* -from *clock* -to *pin*
3. *command* -from *pin* -to *clock*
4. *command* -from *pin*
5. *command* -to *pin*
6. *command* -from *clock* -to *clock*
7. *command* -from *clock*
8. *command* -to *clock*

For example, in the following command sequence, paths from A to B are treated as two-cycle paths because specific commands override general commands:

```
dc_shell> set_multicycle_path 2 -from A -to B
dc_shell> set_multicycle_path 3 -from A
```

The following rules summarize the interaction of the timing exception commands:

- General `set_false_path` commands override specific `set_multicycle_path` commands.
- General `set_max_delay` commands override specific `set_multicycle_path` commands.

- Specific `set_false_path` commands override specific `set_max_delay` or `set_min_delay` commands.
- Specific `set_max_delay` commands override specific `set_multicycle_path` commands.

Setting Area Constraints

The `set_max_area` command specifies the maximum area for the current design by placing a `max_area` attribute on the current design. Specify the area in the same units used for area in the technology library.

For example, to set the maximum area to 100, enter

```
dc_shell> set_max_area 100
```

Design area consists of the areas of each component and net. The following components are ignored when Design Compiler calculates design area:

- Unknown components
- Components with unknown areas
- Technology-independent generic cells

Cell (component) area is technology dependent; Design Compiler obtains this information from the technology library.

When you specify both timing and area constraints, Design Compiler attempts to meet timing goals before area goals. To prioritize area constraints over total negative slack (but not over worst negative slack), use the `-ignore_tns` option when you specify the area constraint.

```
dc_shell> set_max_area -ignore_tns 100
```

To optimize a small area, regardless of timing, remove all constraints except for maximum area. You can use the `remove_constraint` command to remove constraints from your design. Be aware that this command removes *all* optimization constraints from your design.

Verifying the Precompiled Design

Before compiling your design, verify that

- The design is consistent

Use the `check_design` command to verify design consistency. For information about the `check_design` command, see [“Checking for Design Consistency” on page 9-2](#).

- The attributes and constraints are correct

Design Compiler provides many commands for reporting the attributes and constraints. For information about these commands, see [“Analyzing Design Problems” on page 9-8](#) and [“Analyzing Timing Problems” on page 9-9](#).

8

Optimizing the Design

Optimization is the Design Compiler synthesis step that maps the design to an optimal combination of specific target library cells, based on the design's functional, speed, and area requirements. Several of the many factors affecting the optimization outcome are discussed in this chapter.

This chapter has the following sections:

- [The Optimization Process](#)
- [Selecting and Using a Compile Strategy](#)
- [Resolving Multiple Instances of a Design Reference](#)
- [Preserving Subdesigns](#)
- [Understanding the Compile Cost Function](#)
- [Performing Design Exploration](#)

- [Performing Design Implementation](#)
- [Using DC Ultra Datapath Optimization](#)

The Optimization Process

Design Compiler performs the following three levels of optimization:

- Architectural optimization
- Logic-level optimization
- Gate-level optimization

The following sections describe these processes.

Architectural Optimization

Architectural optimization works on the HDL description. It includes such high-level synthesis tasks as

- Sharing common subexpressions
- Sharing resources
- Selecting DesignWare implementations
- Reordering operators
- Identifying arithmetic expressions for data-path synthesis (DC Ultra only).

Except for DesignWare implementations, these high-level synthesis tasks occur only during the optimization of an unmapped design. DesignWare selection can recur after gate-level mapping.

High-level synthesis tasks are based on your constraints and your HDL coding style. After high-level optimization, circuit function is represented by GTECH library parts, that is, by a generic, technology-independent netlist.

For more information about how your coding style affects architectural optimization, see [Chapter 3, “Preparing Design Files for Synthesis”](#).

Logic-Level Optimization

Logic-level optimization works on the GTECH netlist. It consists of the following two processes:

- Structuring

This process adds intermediate variables and logic structure to a design, which can result in reduced design area. Structuring is constraint based. It is best applied to noncritical timing paths.

During structuring, Design Compiler searches for subfunctions that can be factored out and evaluates these factors, based on the size of the factor and the number of times the factor appears in the design. Design Compiler turns the subfunctions that most reduce the logic into intermediate variables and factors them out of the design equations.

By default, Design Compiler structures your design. Use the `set_structure` command and the `compile_new_boolean_structure` variable to control the structuring of your design. The `set_structure` command and its options set the following attributes: `structure`, `structure_boolean`, and `structure_timing`.

- Flattening

The goal of this process is to convert combinational logic paths of the design to a two-level, sum-of-products representation. Flattening is carried out independently of constraints. It is useful for speed optimization because it leads to just two levels of combinational logic.

During flattening, Design Compiler removes all intermediate variables, and therefore all its associated logic structure, from a design. Flattening is not always practical, however, because it requires a large amount of CPU time and can increase area.

By default, Design Compiler does not flatten your design. Use the `set_flatten` command to control flattening of your design. The `set_flatten` command and its options set the following **attributes**: `flatten`, `flatten_effort`, `flatten_minimize`, and `flatten_phase`.

Note:

Flattening does not collapse design hierarchy. In Design Compiler, you remove levels of design hierarchy by using the `ungroup` command or the `compile` command with the `-ungroup_all` or `-auto_ungroup` option.

The structuring and flattening attributes enable fine-tuning of the optimization techniques used for each design in the design hierarchy. [Table 8-1](#) shows the default values for these attributes.

Table 8-1 Structuring and Flattening Attributes

Attribute	Default setting
structure	true
structure_boolean	false
structure_timing	true
flatten	false

Use the `report_compile_options` command to display these attributes for the current design.

Note:

Do not change these default settings unless you understand clearly the impact of structuring and flattening on optimization. If you do choose to change these settings, change them on a design-by-design basis in the hierarchy; do not set these attributes globally. If you specify both flattening and structuring, Design Compiler first performs flattening, then structuring.

Gate-Level Optimization

Gate-level optimization works on the generic netlist created by logic synthesis to produce a technology-specific netlist. It includes the following processes:

- Mapping

This process uses gates (combinational and sequential) from the target technology libraries to generate a gate-level implementation of the design whose goal is to meet timing and area goals. You can use the various `compile` command options to control the mapping algorithms used by Design Compiler.

- Delay optimization

The process goal is to fix delay violations introduced in the mapping phase. Delay optimization does not fix design rule violations or meet area constraints.

- Design rule fixing

The process 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.

- Area optimization

The process goal is to meet area constraints after the mapping, delay optimization, and design rule fixing phases are completed. However, Design Compiler does not allow area recovery to introduce design rule or delay constraint violations as a means of meeting the area constraints.

You can change the priority of the constraints by using the `set_cost_priority` command. Also, you can disable design rule fixing by specifying the `-no_design_rule` option when you run the `compile` command. However, if you use this option, your synthesized design might violate design rules.

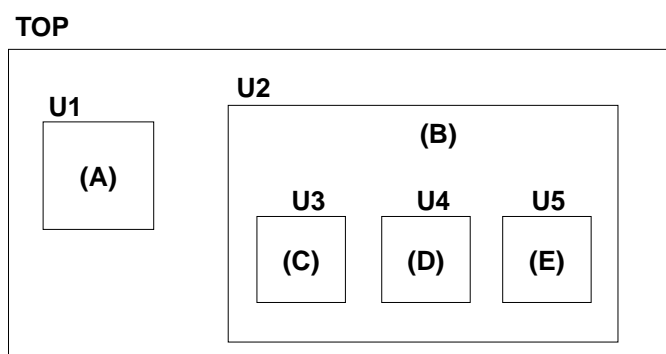
Selecting and Using a Compile Strategy

You can use various strategies to compile your hierarchical design. The basic strategies are

- Top-down compile, in which the top-level design and all its subdesigns are compiled together
- Bottom-up compile, in which the individual subdesigns are compiled separately, starting from the bottom of the hierarchy and proceeding up through the levels of the hierarchy until the top-level design is compiled
- Mixed compile, in which the top-down or bottom-up strategy, whichever is most appropriate, is applied to the individual subdesigns

In the following sections, the top-down and bottom-up compile strategies are demonstrated, using the simple design shown in [Figure 8-1](#).

Figure 8-1 Design to Illustrate Compile Strategies



The top-level, or global, specifications for this design, given in [Table 8-2](#), are defined by the script of [Example 8-1](#) or [Example 8-2](#). These specifications apply to TOP and all its subdesigns.

Table 8-2 Design Specifications for Design TOP

Specification type	Value
Operating condition	WCCOM
Wire load model	"20x20"
Clock frequency	40 MHz
Input delay time	3 ns
Output delay time	2 ns
Input drive strength	drive_of (IV)
Output load	1.5 pF

Example 8-1 dcsh Constraints File for Design TOP (defaults.con)

```
set_operating_conditions WCCOM
set_wire_load_model "20x20"
create_clock -period 25 clk
set_input_delay 3 -clock clk all_inputs()-find(port, clk)
set_output_delay 2 -clock clk all_outputs()
set_load 1.5 all_outputs()
set_driving_cell -cell IV all_inputs()
set_drive 0 clk
```

Example 8-2 dctcl Constraints File for Design TOP (defaults.con)

```
set_operating_conditions WCCOM
set_wire_load_model "20x20"
create_clock -period 25 clk
set_input_delay 3 -clock clk \
    [remove_from_collection [all_inputs] [get_ports clk]]
set_output_delay 2 -clock clk [all_outputs]
set_load 1.5 [all_outputs]
set_driving_cell -cell IV [all_inputs]
set_drive 0 clk
```

Note:

To prevent buffering of the clock network, the script sets the input drive resistance of the clock port (clk) to 0 (infinite drive strength).

Top-Down Compile

You can use the top-down compile strategy for designs that are not memory or CPU limited. Furthermore, top-level designs that are memory limited can often be compiled using the top-down strategy if you first replace some of the subdesigns with interface logic model representations. Replacing a subdesign with an interface logic model can greatly reduce the memory requirements for the subdesign instantiation in the top-level design. For information about

how to generate and use interface logic models, see [Chapter 10, “Creating, Instantiating, and Using Interface Logic Models in Hierarchical Synthesis.”](#)

The top-down compile strategy has these advantages:

- Provides a push-button approach
- Takes care of interblock dependencies automatically

On the other hand, the top-down compile strategy requires more memory and might result in longer runtimes for designs with over 100K gates.

To implement a top-down compile, carry out the following steps:

Note:

If your top-level design contains one or more interface logic models, use the compile flow described in [Chapter 10, “Creating, Instantiating, and Using Interface Logic Models in Hierarchical Synthesis.”](#)

1. Read in the entire design.
2. Resolve multiple instances of any design references.

A design that is referenced by more than one instantiated block or cell must be resolved. Otherwise, Design Compiler cannot compute which environmental attributes and constraints to apply to the design during optimization. To learn how to deal with this problem, see [“Resolving Multiple Instances of a Design Reference”](#) on page 8-20.

3. Apply attributes and constraints to the top level.

Attributes and constraints implement the design specification. For information about attributes, see [“Working With Attributes” on page 5-50](#). For information about constraints, see [Chapter 6, “Defining the Design Environment”](#) and [Chapter 7, “Defining Design Constraints.”](#)

Note:

You can assign local attributes and constraints to subdesigns, provided that those attributes and constraints are defined with respect to the top-level design.

4. Compile the design.

A top-down compile script for the TOP design is shown in [Example 8-3](#) (dcsh mode) and [Example 8-4](#) (dctcl mode). Both scripts contain comments that identify each of the steps. The constraints are applied by including the constraint file (defaults.con) shown in [Example 8-2 on page 8-9](#).

Example 8-3 Top-Down Compile Script (dcsh Mode)

```
/* read in the entire design */
read_file -format verilog E.v
read_file -format verilog D.v
read_file -format verilog C.v
read_file -format verilog B.v
read_file -format verilog A.v
read_file -format verilog TOP.v
current_design TOP
link

/* apply constraints and attributes */
include defaults.con

/* compile the design */
compile
```

Example 8-4 Top-Down Compile Script (dctcl Mode)

```
/* read in the entire design */
read_verilog E.v
read_verilog D.v
read_verilog C.v
read_verilog B.v
read_verilog A.v
read_verilog TOP.v
current_design TOP
link

/* apply constraints and attributes */
source defaults.con

/* compile the design */
compile
```

Bottom-Up Compile

Use the bottom-up compile strategy for medium-size and large designs.

Note:

The bottom-up compile strategy is also known as the compile-characterize-write_script-recompile method.

The bottom-up compile strategy provides these advantages:

- Compiles large designs by using the divide-and-conquer approach
- Requires less memory than top-down compile
- Allows time budgeting

The bottom-up compile strategy requires

- Iterating until the interfaces are stable

- Manual revision control

The bottom-up compile strategy compiles the subdesigns separately and then incorporates them in the top-level design. The top-level constraints are applied, and the design is checked for violations. Although it is possible that no violations are present, this outcome is unlikely because the interface settings between subdesigns usually are not sufficiently accurate at the start.

To improve the accuracy of the interblock constraints, you read in the top-level design and all compiled subdesigns and apply the `characterize` command to the individual cell instances of the subdesigns. Based on the more realistic environment provided by the compiled subdesigns, `characterize` captures environment and timing information for each cell instance and then replaces the existing attributes and constraints of each cell's referenced subdesign with the new values.

Using the improved interblock constraint, you recompile the characterized subdesigns and again check the top-level design for constraint violations. You should see improved results, but you might need to iterate the entire process several times to remove all significant violations.

The bottom-up compile strategy requires these steps:

1. Develop both a default constraint file and subdesign-specific constraint files.

The default constraint file includes global constraints, such as the clock information and the drive and load estimates. The subdesign-specific constraint files reflect the time budget allocated to the subblocks.

2. Compile the subdesigns independently.

3. Read in the top-level design and any compiled subdesigns not already in memory.
4. Set the current design to the top-level design, link the design, and apply the top-level constraints.

If the design meets its constraints, you are finished. Otherwise, continue with the following steps.

5. Apply the `characterize` command to the cell instance with the worst violations.
6. Use `write_script` to save the characterized information for the cell.

You use this script to re-create the new attribute values when you are recompiling the cell's referenced subdesign.

7. Use `remove_design -all` to remove all designs from memory.
8. Read in the RTL design of the previously characterized cell.

Recompiling the RTL design instead of the cell's mapped design usually leads to better optimization.

9. Set `current_design` to the characterized cell's subdesign and recompile, using the saved script of characterization data.
10. Read in all other compiled subdesigns.
11. Link the current subdesign.
12. Choose another subdesign, and repeat steps 3 through 9 until you have recompiled all subdesigns, using their actual environments.

When applying the bottom-up compile strategy, consider the following:

- The `read_file` command runs most quickly with the `.db` format. If you will not be modifying your RTL code after the first time you read (or elaborate) it, save the unmapped design to a `.db` file. This will save time when you reread the design.
- The `compile` command affects all subdesigns of the current design. If you want to optimize only the current design, you can remove or not include its subdesigns in your database, or you can place the `dont_touch` attribute on the subdesigns (by using the `set_dont_touch` command).

A bottom-up compile script for the TOP design is shown in [Example 8-5 on page 8-16](#) (dcsh mode) and [Example 8-6 on page 8-18](#) (dctl mode). Both scripts contain comments that identify each of the steps in the bottom-up compile strategy. In these scripts it is assumed that block constraint files exist for each of the subblocks (subdesigns) in design TOP. The compile scripts also use the default constraint file (defaults.con) shown in [Example 8-2 on page 8-9](#).

Note:

This script shows only one pass through the bottom-up compile procedure. If the design requires further compilations, you repeat the procedure from the point where the top-level design, TOP.v, is read in.

Example 8-5 Bottom-Up Compile Script (dcsh Mode)

```
all_blocks = {E,D,C,B,A}

/* compile each subblock independently */
foreach (block, all_blocks) {
    /* read in block */
    block_source = block + ".v"
    read_file -format verilog block_source
    current_design block
    link
    /* apply global attributes and constraints */
    include defaults.con
    /* apply block attributes and constraints */
    block_script = block + ".con"
    include block_script
    /* compile the block */
    compile
}

/* read in entire compiled design */
read_file -format verilog TOP.v
current_design TOP
link
write -hierarchy -output first_pass.db

/* apply top-level constraints */
include defaults.con
include top_level.con

/* check for violations */
report_constraint

/* characterize all instances in the design */
all_instances = {U1,U2,U2/U3,U2/U4,U2/U5}
characterize -constraint all_instances

/* save characterize information */
foreach (block, all_blocks) {
    current_design block
    char_block_script = block + ".wscr"
    write_script > char_block_script
}

/* recompile each block */
```

```

foreach (block, all_blocks) {

    /* clear memory */
    remove_design -all

    /* read in previously characterized subblock */
    block_source = block + ".v"
    read_file -format verilog block_source

    /* recompile subblock */
    current_design block
    link
    /* apply global attributes and constraints */
    include defaults.con
    /* apply characterization constraints */
    char_block_script = block + ".wscr"
    include char_block_script
    /* apply block attributes and constraints */
    block_script = block + ".con"
    include block_script
    /* recompile the block */
    compile
}

```

Example 8-6 Bottom-Up Compile Script (dctcl Mode)

```
set all_blocks {E D C B A}

# compile each subblock independently
foreach block $all_blocks {
    # read in block
    set block_source "$block.v"
    read_file -format verilog $block_source
    current_design $block
    link
    # apply global attributes and constraints
    source defaults.con
    # apply block attributes and constraints
    set block_script "$block.con"
    source $block_script
    # compile the block
    compile
}

# read in entire compiled design
read_file -format verilog TOP.v
current_design TOP
link
write -hierarchy -output first_pass.db

# apply top-level constraints
source defaults.con
source top_level.con

# check for violations
report_constraint

# characterize all instances in the design
set all_instances {U1 U2 U2/U3 U2/U4 U2/U5}
characterize -constraint $all_instances

# save characterize information
foreach block $all_blocks {
    current_design $block
    set char_block_script "$block.wscr"
    write_script > $char_block_script
}

# recompile each block
```

```

foreach block $all_blocks {

    # clear memory
    remove_design -all

    # read in previously characterized subblock
    set block_source "$block.v"
    read_file -format verilog $block_source

    # recompile subblock
    current_design $block
    link
    # apply global attributes and constraints
    source defaults.con
    # apply characterization constraints
    set char_block_script "$block.wscr"
    source $char_block_script
    # apply block attributes and constraints
    set block_script "$block.con"
    source $block_script
    # recompile the block
    compile
}

```

Note:

When performing a bottom-up compile, if the top-level design contains glue logic as well as the subblocks (subdesigns), you must also compile the top-level design. In this case, to prevent Design Compiler from recompiling the subblocks, you first apply the `set_dont_touch` command to each subdesign.

Mixed Compile Strategy

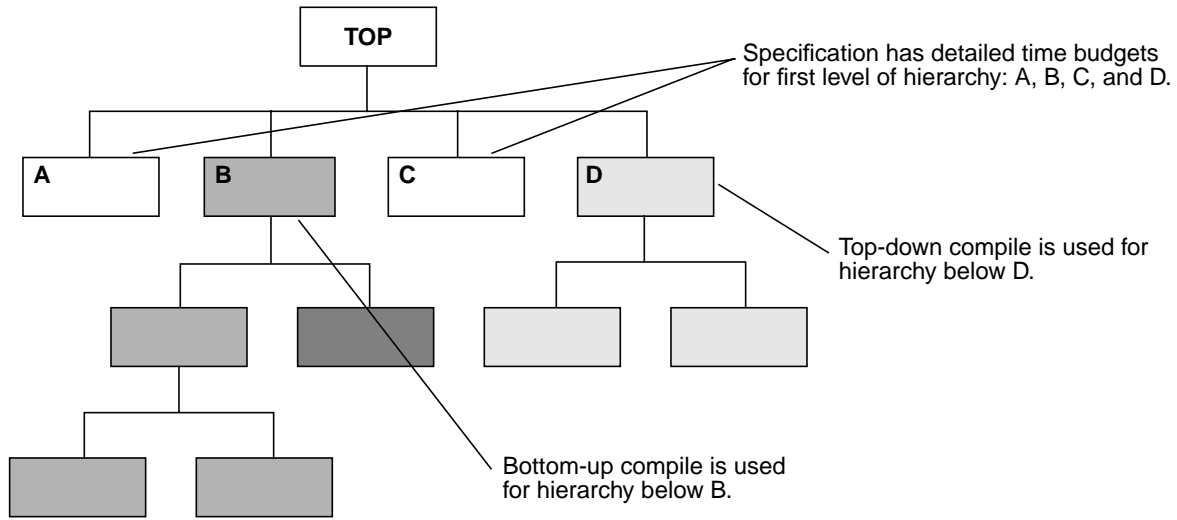
You can take advantage of the benefits of both the top-down and the bottom-up compile strategies by using both.

- Use the top-down compile strategy for small hierarchies of blocks.

- Use the bottom-up compile strategy to tie small hierarchies together into larger blocks.

Figure 8-2 shows an example of the mixed compilation strategy.

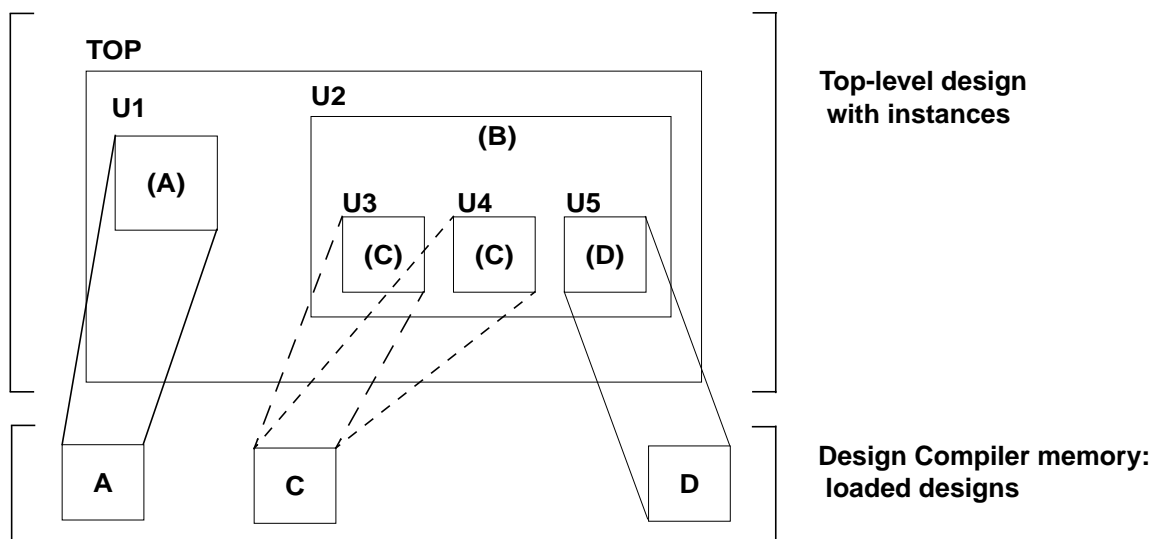
Figure 8-2 Mixing Compilation Strategies



Resolving Multiple Instances of a Design Reference

In a hierarchical design, subdesigns are often referenced by more than one cell instance, that is, multiple references of the design can occur. For example, Figure 8-3 shows the design TOP, in which design C is referenced twice (U2/U3 and U2/U4).

Figure 8-3 Multiple Instances of a Design Reference



The following methods are available for handling designs with multiple instances:

- The `uniquify` method

In earlier releases, you had manually to run the `uniquify` command to create a uniquely named copy of the design for each instance. However, beginning with version V-2004.06, the tool automatically uniquifies 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 runtimes because the tool automatically “re-uniquifies” the designs when you run the `compile` command. You cannot turn off the uniquify process.

- The compile-once-don't-touch method

This method uses the `set_dont_touch` command to preserve the compiled subdesign while the remaining designs are compiled.

- The ungroup method

This method uses the `ungroup` command to remove the hierarchy.

These methods are described in the following sections.

Uniquify Method

The uniquify process copies and renames any multiply referenced design so that each instance references a unique design. This process can be invoked manually by running the `uniquify` command or automatically when you run the `compile` command. The uniquification process can resolve multiple references throughout the hierarchy the current design (except those having a `dont_touch` attribute). After this process completes, the tool can optimize each design copy based on the unique environment of its cell instance.

You can also create unique copies for specific references by using the `-reference` option, or you can specify specific cells by using the `-cells` option. Design Compiler makes unique copies for cells specified with the `-reference` or the `-cells` option, even if they have a `dont_touch` attribute.

Design Compiler uses the naming convention specified in the `uniquify_naming_style` variable to generate the name for each copy of the subdesign. The default naming convention is

`%s_%d`

`%s`

The original name of the subdesign (or the name specified in the `-base_name` option).

`%d`

The smallest integer value that forms a unique subdesign name.

You can use the `uniquify` command simply to resolve multiple design references, or you can recompile the current design after the multiple references to the subdesigns are resolved.

Note:

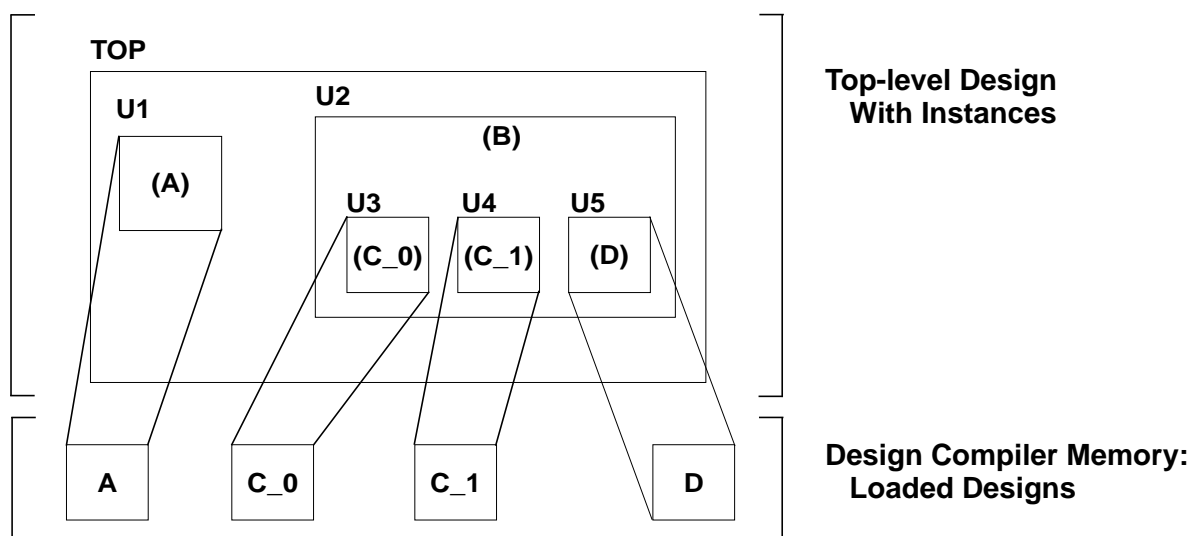
Without recompiling the design, you can use the `uniquify` command together with the `uniquify_naming_style` variable simply to resolve multiple design references in a netlist.

The following command sequence resolves the multiple instances of design C in design TOP shown in [Figure 8-3 on page 8-21](#); it uses the automatic `uniquify` method to create new designs C_0 and C_1 by copying design C and then replaces design C with the two copies in memory.

```
dc_shell> current_design top  
dc_shell> compile
```

[Figure 8-4](#) shows the result of running this command sequence.

Figure 8-4 Uniquify Results



Compared with the compile-once-don't-touch method, the uniquify method has the following characteristics:

- Requires more memory
- Takes longer to compile

Compile-Once-Don't-Touch Method

If the environments around the instances of a multiply referenced design are sufficiently similar, use the compile-once-don't-touch method. In this method, you compile the design, using the environment of one of its instances, and then you use the `set_dont_touch` command to preserve the subdesign during the remaining optimization. For details about the `set_dont_touch` command, see [“Preserving Subdesigns” on page 8-28](#).

To use the compile-once-don't-touch method to resolve multiple instances, follow these steps:

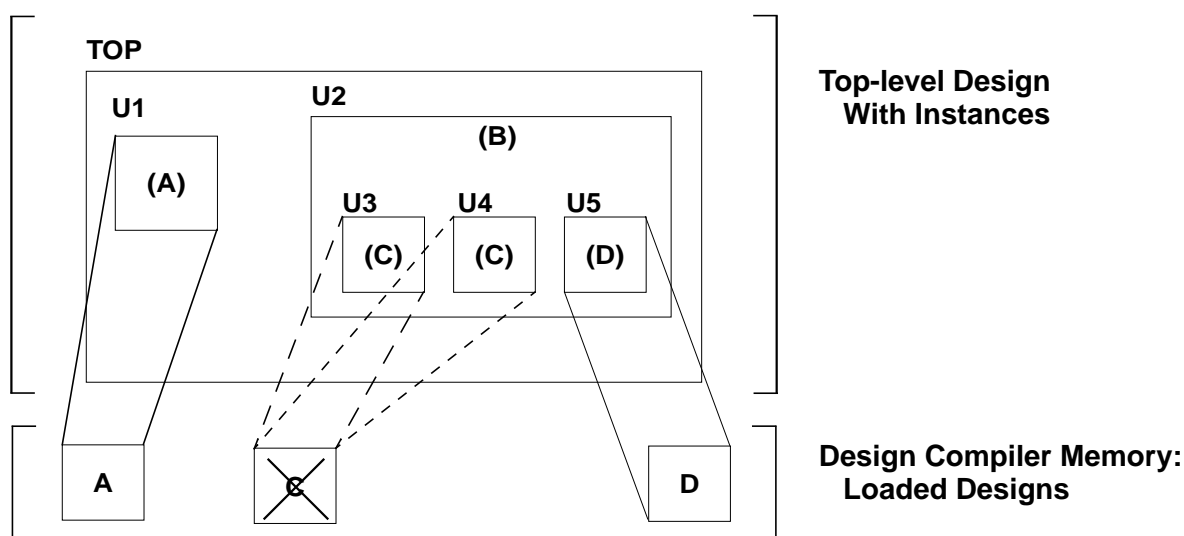
1. Characterize the subdesign's instance that has the worst-case environment.
2. Compile the referenced subdesign.
3. Use the `set_dont_touch` command to set the `dont_touch` attribute on all instances that reference the compiled subdesign.
4. Compile the entire design.

For example, the following command sequence resolves the multiple instances of design C in design TOP by using the compile-once-don't-touch method (assuming U2/U3 has the worst-case environment). In this case, no copies of the original subdesign are loaded into memory.

```
dc_shell> current_design top
dc_shell> characterize U2/U3
dc_shell> current_design C
dc_shell> compile
dc_shell> current_design top
dc_shell> set_dont_touch {U2/U3 U2/U4}
dc_shell> compile
```

Figure 8-5 shows the result of running this command sequence. The X drawn over the C design, which has already been compiled, indicates that the `dont_touch` attribute has been set. This design is not modified when the top-level design is compiled.

Figure 8-5 Compile-Once-Don't-Touch Results



The compile-once-don't-touch method has the following advantages:

- Compiles the reference design once
- Requires less memory than the uniquify method
- Takes less time to compile than the uniquify method

The principal disadvantage of the compile-once-don't-touch method is that the characterization might not apply well to all instances. Another disadvantage is that you cannot ungroup objects that have the `dont_touch` attribute.

Ungroup Method

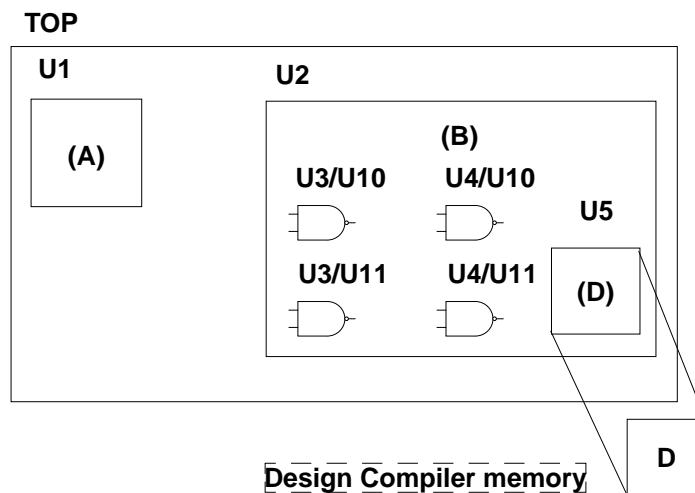
The ungroup method has the same effect as the uniquify method (it makes unique copies of the design), but in addition, it removes levels of hierarchy. This method uses the `ungroup` command to produce a flattened netlist. For details about the `ungroup` command, See [“Removing Levels of Hierarchy” on page 5-27](#).

After ungrouping the instances of a subdesign, you can recompile the top-level design. For example, the following command sequence uses the ungroup method to resolve the multiple instances of design C in design TOP:

```
dc_shell> current_design B  
dc_shell> ungroup {U3 U4}  
dc_shell> current_design top  
dc_shell> compile
```

Figure 8-6 shows the result of running this command sequence.

Figure 8-6 Ungroup Results



The ungroup method has the following characteristics:

- Requires more memory and takes longer to compile than the compile-once-don't-touch method
- Provides the best synthesis results

The obvious drawback in using the ungroup method is that it removes the user-defined design hierarchy.

Preserving Subdesigns

The `set_dont_touch` command preserves a subdesign during optimization. It places the `dont_touch` attribute on cells, nets, references, and designs in the current design to prevent these objects from being modified or replaced during optimization.

Note:

Any interface logic model present in your design is automatically marked as `dont_touch`. Also, the cells of an interface logic model are marked as `dont_touch`. For information about interface logic models, see [Chapter 10, “Creating, Instantiating, and Using Interface Logic Models in Hierarchical Synthesis.”](#)

Use the `set_dont_touch` command on subdesigns you do not want optimized with the rest of the design hierarchy. The `dont_touch` attribute does not prevent or disable timing through the design.

When you use `set_dont_touch`, remember the following points:

- Setting `dont_touch` on a hierarchical cell sets an implicit `dont_touch` on all cells below that cell.
- Setting `dont_touch` on a library cell sets an implicit `dont_touch` on all instances of that cell.
- Setting `dont_touch` on a net sets an implicit `dont_touch` only on mapped combinational cells connected to that net. If the net is connected only to generic logic, optimization might remove the net.
- Setting `dont_touch` on a reference sets an implicit `dont_touch` on all cells using that reference during subsequent optimizations of the design.

- Setting `dont_touch` on a design has an effect only when the design is instantiated within another design as a level of hierarchy. In this case, the `dont_touch` attribute on the design implies that all cells under that level of hierarchy are subject to the `dont_touch` attribute. Setting `dont_touch` on the top-level design has no effect because the top-level design is not instantiated within any other design.
- You cannot manually or automatically ungroup objects marked as `dont_touch`. That is, the `ungroup` command and the `compile -ungroup_all` and `-auto_ungroup` options have no effect on `dont_touch` objects.

Note:

The `dont_touch` attribute is ignored on synthetic part cells (for example, many of the cells read in from an HDL description) and on nets that have unmapped cells on them. During compilation, warnings appear for `dont_touch` nets connected to unmapped cells (generic logic).

Use the `report_design` command to determine whether a design has the `dont_touch` attribute set.

```
dc_shell> set_dont_touch SUB_A
Performing set_dont_touch on design 'SUB_A'.
1
dc_shell> report_design
```

```
*****
Report : design
Design : SUB_A
Version: 1999.05
Date   : Mon Jan 4 10:56:49 1999
*****
```

Design is dont_touched.

To remove the `dont_touch` attribute, use the `remove_attribute` command or the `set_dont_touch` command set to false.

Understanding the Compile Cost Function

The compile cost function consists of design rule costs and optimization costs. By default, Design Compiler prioritizes costs in the following order:

1. Design rule costs
 - a. Connection class
 - b. Multiple port nets
 - c. Maximum transition time
 - d. Maximum fanout
 - e. Maximum capacitance
 - f. Cell degradation
2. Optimization costs
 - a. Maximum delay
 - b. Minimum delay
 - c. Maximum power
 - d. Maximum area
 - e. Minimum porosity

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.

When evaluating cost function components, Design Compiler considers only violators (positive difference between actual value and constraint) and works to reduce the cost function to zero.

The goal of Design Compiler is to meet all constraints. However, by default, it gives precedence to design rule constraints because design rule constraints are functional requirements for designs. Using the default priority, Design Compiler fixes design rule violations even at the expense of violating your delay or area constraints.

You can change the priority of the maximum design rule costs and the delay costs by using the `set_cost_priority` command to specify the ordering. You must run the `set_cost_priority` command before running the `compile` command.

You can disable evaluation of the design rule cost function by using the `-no_design_rule` option when running the `compile` command.

You can disable evaluation of the optimization cost function by using the `-only_design_rule` option when running the `compile` command.

Calculating Transition Time Cost

Design Compiler determines driver transition times from the technology 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

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

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.

Calculating Cell Degradation Cost

The cell degradation tables in the technology library provide a secondary maximum capacitance constraint, based on the transition times at the cell inputs. Design Compiler evaluates this cost only if you set the `compile_fix_cell_degradation` variable to true.

If the `compile_fix_cell_degradation` variable is true and the calculated capacitance is greater than the `cell_degradation` value, Design Compiler reports a design rule violation and attempts to correct the violation. The maximum capacitance cost has a higher priority than the cell degradation cost. Therefore, Design Compiler fixes cell degradation violations only if it can do so without violating the maximum capacitance constraint.

Calculating Maximum Delay Cost

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 (see [“Creating Path Groups” on page 8-45](#) for information about 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.

Design Compiler calculates the maximum delay violation for each path group as

$$\max (0, (\text{actual_path_delay} - \text{max_delay}))$$

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.

For information about specifying the critical range, see [“Creating Path Groups” on page 8-45](#).

Using the critical negative slack method, the equation for the maximum delay cost is

$$\sum_{i=1}^m \left(\left\langle \sum_{j=1}^n v_{ij} \right\rangle \times w_i \right)$$

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, (min_delay - 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 technology 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 Reference Manual*.

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 technology library.

Design Compiler calculates the maximum area cost as

```
max (0, actual_area - max_area)
```

Calculating Minimum Porosity Cost

Design Compiler computes the porosity of a design by dividing the sum of the routing track area of each of its components on the design hierarchy's lowest level by the sum of all component areas. Design Compiler ignores the following components when calculating porosity:

- Unknown components
- Components with unknown routing track areas
- Technology-independent generic cells

The routing track area of a cell and the cell area are technology dependent. Design Compiler obtains this information from the technology library.

Design Compiler calculates the minimum porosity cost as

```
max (0, min_porosity - actual_porosity)
```

Performing Design Exploration

In Design exploration, you use the default synthesis algorithm to gauge the design performance against your goals. To invoke the default synthesis algorithm, use the `compile` command with no options:

```
dc_shell> compile
```

The default compile uses the `-map_effort medium` option of the `compile` command and the default settings of the structuring and flattening attributes. The default area effort of the area recovery phase of the compile is the specified value of the `map_effort` option. You can change the area effort by using the `-area_effort` option.

If the performance violates the timing goals by more than 15 percent, you should consider whether to refine the design budget or modify the HDL code.

Performing Design Implementation

The default compile generates good results for most designs. If your design meets the optimization goals after design exploration, you are finished. If not, try the techniques described in the following sections:

- [Optimizing Random Logic](#)
- [Optimizing Structured Logic](#)
- [Optimizing High-Speed Designs](#)
- [Optimizing for Maximum Performance](#)
- [Optimizing for Minimum Area](#)
- [Optimizing Data Paths](#)

Optimizing Random Logic

If the default compile does not give the desired result for your random logic design, try the following techniques. If the first technique does not give the desired results, use the second technique, and so on, until you obtain the desired results.

- Flatten the design before structuring. Enter

```
dc_shell> set_flatten true  
dc_shell> set_structure true  
dc_shell> compile
```

When you run this command sequence, Design Compiler first flattens the logic, then goes back and restructures the design by sharing logic off the critical path.

- Increase the flattening effort. Enter

```
dc_shell> set_flatten true -effort medium
dc_shell> compile
```

- Fine-tune the results with minimization or phase inversion. Enter

```
dc_shell> set_flatten true \
           -minimize multiple_output -phase true
dc_shell> compile
```

The `set_flatten -minimize` command causes Design Compiler to share product terms between output logic cones (minimization). Minimization causes higher fanout but does not change the two-level sum-of-products representation.

If you select the `-minimize single_output` option, Design Compiler minimizes the equations for each output individually. The `-minimize multiple_output` option enables minimization of the entire design by allowing optimization to share terms among outputs. Minimization increases compile time; therefore, Design Compiler does not perform minimization during default flattening.

The `set_flatten -phase true` command inverts the polarity of the outputs, compares the original implementation with the complement, and keeps the best result. Setting the `-phase` option to true increases compile time; therefore, the default value for the `-phase` option is false.

Optimizing Structured Logic

If the default compile does not give the desired result for your structured logic design, try the following techniques. If the first technique does not give the desired results, try the second one.

- Map the design with no flattening or structuring. Enter

```
dc_shell> set_structure false  
dc_shell> compile
```

- Flatten with structuring. Enter

```
dc_shell> set_flatten true  
dc_shell> set_structure true  
dc_shell> compile
```

When you run this command sequence, Design Compiler first flattens the logic, then goes back and restructures the design by sharing logic off the critical path.

Optimizing High-Speed Designs

For high-speed designs that are timing critical, you can invoke a single DC Ultra command, `compile_ultra`, for critical delay optimization. This command allows you to apply the best possible set of timing-centric variables or commands during compile so that delay QOR is significantly improved. Because this command includes all compile options and starts the entire compile process, no separate `compile` command is necessary.

Note:

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

The command syntax is

```
int compile_ultra [-scan] [-no_uniquify] [-no_autoungroup]
                 [-no_boundary_optimization]
```

`-scan`

Enables test-ready compile. This option replaces all sequential elements with their scan-equivalent cells.

`-no_uniquify`

If this option is specified, the uniquify process is not performed. Use this option when you want to carry out a first-pass, bottom-up, high-effort compile.

`-no_autoungroup`

Allows you to specify that no auto-ungrouping be done during compile.

If this option is not enabled, Design Compiler performs delay-based auto-ungrouping by default. Delay-based auto-ungrouping ungroups hierarchies along the critical path and is used essentially for timing optimization. You define the maximum size of the hierarchy to be ungrouped by setting the `compile_auto_ungroup_delay_num_cells` variable. The default is 500. Note that the number-of-cells limit of a hierarchy refers to its child cells.

`-no_boundary_optimization`

Lets you specify that no hierarchical boundary optimization is performed on the current design.

By default, if this option is not specified, hierarchical boundary optimization is carried out, which can change design function such that the design only operates properly in its current environment. If this behavior is undesirable, use the option to disable boundary optimization.

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).

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

For more information on this command, see the man page.

Optimizing for Maximum Performance

If your design does not meet the timing constraints, you can try the following methods to improve performance:

- Create path groups
- Fix heavily loaded nets
- Flatten logic on the critical path
- Auto-ungroup hierarchies on the critical path
- Perform a high-effort incremental compile
- Perform a high-effort compile

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.

Use the `group_path` command to create path groups. The `group_path` command allows you to

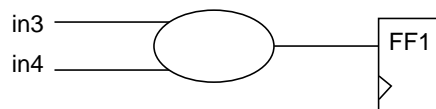
- Control the optimization of your design
- Optimize near-critical paths
- Optimize all paths

Controlling the Optimization of 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 8-7](#) shows a design that has multiple paths to flip-flop FF1.

Figure 8-7 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.

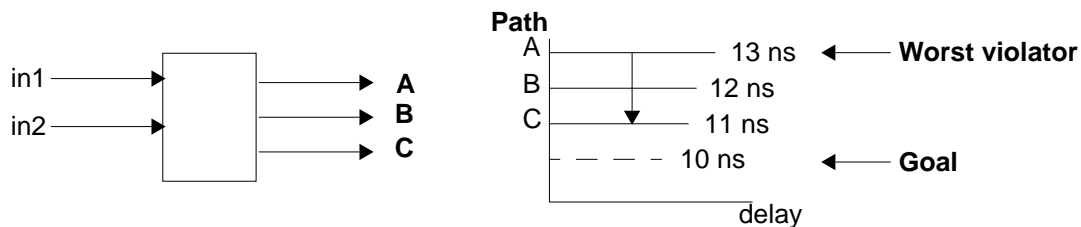
Specifying a critical range can increase runtime. To limit the runtime increase, use critical range only during the final implementation phase of the design, and use a reasonable critical range value. A guideline for the maximum critical range value is 10 percent of the clock period.

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 8-8](#) shows a design with three outputs, A, B, and C.

Figure 8-8 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. [Table 8-3](#) shows the dcsh and dctcl command sequences for this example.

Table 8-3 Critical Range Examples

dcsh Example	dctcl Example
<pre>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}</pre>	<pre>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}</pre>

Optimizing All Paths. You can optimize all paths by creating a path group for each endpoint in the design. Creating a path group for each endpoint enables total negative slack optimization but results in long compile runtimes.

[Table 8-4](#) shows dcsh and dctcl scripts that you can use to create a path group for each endpoint.

Table 8-4 Scripts to Create a Path Group for Each Endpoint

dcsh Script	dctcl Script
<pre>endpoints = \ all_outputs() + \ all_registers(-data_pins) foreach (endpt, endpoints) { group_path -name endpt -to endpt }</pre>	<pre>set endpoints [add_to_collection \ [all_outputs] \ [all_registers -data_pins]] foreach_in_collection endpt \$endpoints { set pin [get_object_name \$endpt] group_path -name \$pin -to \$pin }</pre>

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. [Table 8-5](#) shows `dcsh` and `dctcl` scripts that use the `balance_buffer` command to fix heavily loaded nets.

Table 8-5 Using `balance_buffer` to Fix Heavily Loaded Nets

dcsh Script	dctcl Script
<pre>include constraints.con compile balance_buffer \ -from find(pin, buf1/Z)</pre>	<pre>source constraints.con compile balance_buffer \ -from [get_pins buf1/Z]</pre>

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. [Table 8-6](#) shows `dcsh` and `dctcl` scripts that use design rules to fix heavily loaded nets.

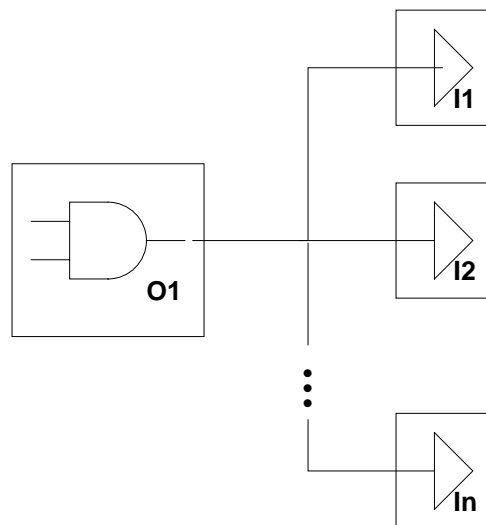
Table 8-6 Using Design Rules to Fix Heavily Loaded Nets

dcsh Script	dctcl Script
<pre>include constraints.con compile set_max_capacitance 3.0 compile -only_design_rule</pre>	<pre>source constraints.con compile set_max_capacitance 3.0 compile -only_design_rule</pre>

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

In the sample design shown in [Figure 8-9](#), 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 8-9 Heavily Loaded Net



Flattening Logic on the Critical Path

Flattening improves performance by representing the design as a two-level sum of products. However, flattening requires considerable CPU resources, and it might not be possible to flatten the entire design. In this case, you can improve the performance by flattening just the logic on the critical path.

To flatten the logic on the critical path,

1. Identify the logic associated with the critical path, using the `all_fanin` command.

2. Group the critical path logic.

Note:

The `group` command groups cells only in the context of the current design; hence you cannot use this flow if the critical path spans across hierarchies.

3. Characterize the critical path logic.
4. Flatten the critical path logic.
5. Ungroup the block of critical path logic.

[Table 8-7](#) shows examples of this procedure using both `dcsh` and `dctcl` syntax.

Table 8-7 *Examples of Flattening Critical Path Logic*

dcsh Example	dctcl Example
<pre>all_fanin -to all_critical_pins() \ -only_cells cp_logic = dc_shell_status group -design critical_block \ -cell_name cp1 cp_logic characterize cp1 current_design critical_block set_flatten true compile set_flatten false current_design .. ungroup -simple_names cp1</pre>	<pre>set cp_logic [all_fanin \ -to [all_critical_pins] -only_cells] group -design critical_block \ -cell_name cp1 \$cp_logic characterize cp1 current_design critical_block set_flatten true compile set_flatten false current_design .. ungroup -simple_names cp1</pre>

Automatically Ungrouping Hierarchies on the Critical Path

Automatically ungrouping hierarchies during compile can often improve performance. Ungrouping removes hierarchy boundaries and allows Design Compiler to optimize over a larger number of gates, generally improving timing. You use delay-based auto-ungrouping to ungroup hierarchies along the critical path.

To use the auto-ungroup capability, you must use the `compile` command option `-auto_ungroup delay`.

For more information on auto-ungrouping, See [“Ungrouping Hierarchies Automatically During Optimization”](#) on page 5-31.

Performing a High-Effort Compile

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).

```
dc_shell> elaborate my_design
dc_shell> compile -map_effort high
```

A high-effort compile pushes Design Compiler to the extreme to achieve the design goal. If you have a DC-Expert license, 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.

```
dc_shell> dont_touch noncritical_blocks
dc_shell> compile -map_effort high -incremental_mapping
```

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

Optimizing for Minimum Area

If your design has timing constraints, these constraints always take precedence over area requirements. For area-critical designs, do not apply timing constraints before you compile. If you want to view timing reports, you can apply timing constraints to the design after you compile.

If your design does not meet the area constraints, you can try the following methods to reduce the area:

- Disable total negative slack optimization
- Enable sequential area recovery
- Enable Boolean optimization
- Manage resource selection
- Use flattening
- Optimize across hierarchical boundaries

Disabling Total Negative Slack Optimization

By default, Design Compiler prioritizes total negative slack over meeting area constraints. This means Design Compiler performs area optimization only on those paths that have positive slack.

To change the default priorities (prioritize area over total negative slack), use the `-ignore_tns` option when setting the area constraints.

```
dc_shell> set_max_area -ignore_tns max_area
```

Enabling Sequential Area Recovery

By default, Design Compiler does not remap sequential elements during optimization. You might be able to reduce area by remapping the sequential elements that are not on the critical path. To enable this capability, set the `compile_sequential_area_recovery` variable to true. You must set this variable before you compile. To set the variable, enter one of the following commands (depending on your shell mode):

```
dc_shell> compile_sequential_area_recovery = true
```

```
dc_shell-t> set compile_sequential_area_recovery true
```

Enabling Boolean Optimization

Boolean optimization uses algorithms based on the basic rules of Boolean algebra. Boolean optimization can use don't care conditions to minimize area. This algorithm performs area optimization only; do not use Boolean optimization for timing-critical designs.

Use the `compile_new_boolean_structure` variable and the `-boolean true` option of the `set_structure` command to enable Boolean optimization. [Table 8-8](#) shows the commands you must run before you compile in both `dcsh` and `dctcl` syntax.

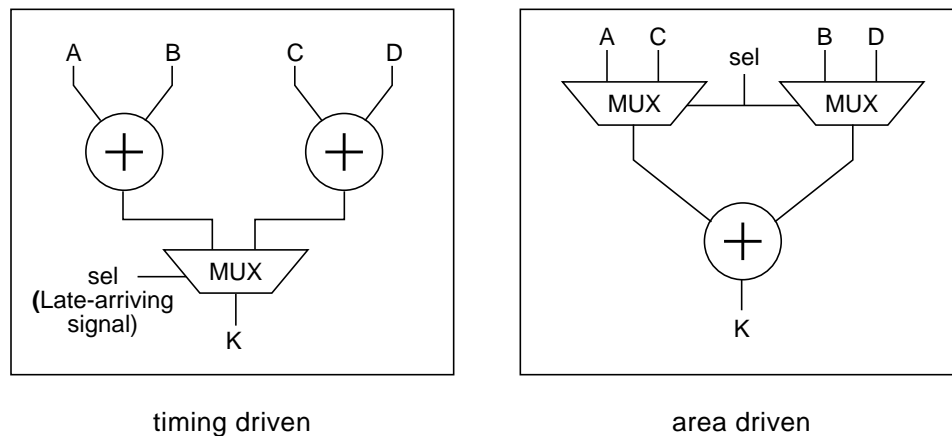
Table 8-8 Examples of Enabling Boolean Optimization

dcsh Example	dctcl Example
<pre>compile_new_boolean_structure = true set_structure true -boolean true \ -boolean_effort medium</pre>	<pre>set compile_new_boolean_structure \ true set_structure true -boolean true \ -boolean_effort medium</pre>

Managing Resource Selection

The decisions made during resource sharing can also significantly affect area. [Figure 8-10](#) shows that high-level optimization can allocate the arithmetic operators in the same HDL code in two very different ways.

Figure 8-10 Resource Sharing Possibilities



The operator implementation also affects area. For example, in [Figure 8-10](#), the timing-driven version implements the adders as carry-lookahead adders, and the area-driven example uses a ripple adder implementation.

By default, high-level optimization performs resource allocation and implementation based on timing constraints. To change the default and force Design Compiler to base resource allocation and implementation on area constraints, set the following variables before compile:

```
dc_shell> set_resource_allocation area_only
dc_shell> set_resource_implementation area_only
```

To specify area-driven resource allocation and implementation for a specific design, set the following variables before you compile:

```
dc_shell> current_design subdesign  
dc_shell> set_resource_allocation area_only  
dc_shell> set_resource_implementation area_only
```

Using Flattening

In most cases, flattening increases the area. In highly random designs with unpredictable structures, flattening might reduce the area. However, flattening is CPU intensive, and the process might not finish for some designs.

Use the `set_flatten` command on specific modules that might benefit from this technique; do not use the `set_flatten` command on the top-level design.

The `-minimize` and `-phase` options, discussed in [“Optimizing Random Logic” on page 8-40](#), can also reduce area.

Optimizing Across Hierarchical Boundaries

Design Compiler respects levels of hierarchy and port functionality (except when automatic ungrouping of small hierarchies is enabled). Boundary optimizations, such as constant propagation through a subdesign, do not occur automatically.

To fine-tune the area, you can leave the hierarchy intact and enable boundary optimization. For greater area reduction, you might have to remove hierarchical boundaries.

Boundary Optimization. Direct Design Compiler to perform optimization across hierarchical boundaries (boundary optimization) by using one of the following commands:

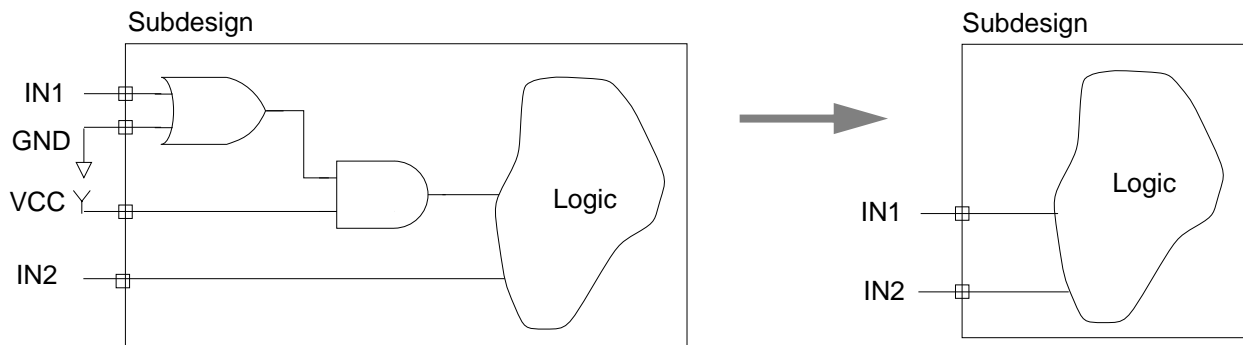
```
dc_shell> compile -boundary_optimization
```

or

```
dc_shell> set_boundary_optimization subdesign
```

If you enable boundary optimization, Design Compiler propagates constants, unconnected pins, and complement information. In designs that have many constants (VCC and GND) connected to the inputs of subdesigns, propagation can reduce area. [Figure 8-11](#) shows this relationship.

Figure 8-11 Benefits of Boundary Optimization



Hierarchy Removal. Removing levels of hierarchy by ungrouping gives Design Compiler more freedom to share common terms across the entire design. You can ungroup specific hierarchies before optimization by using the `set_ungroup` command or the `compile` command with the `-ungroup_all` option to designate which cells you want ungrouped. Also, you can use the auto-ungroup capability of Design Compiler to ungroup small hierarchies during optimization. In this case, you do not specify the hierarchies to be ungrouped.

For details about ungrouping hierarchies, see [“Removing Levels of Hierarchy” on page 5-27](#).

Optimizing Data Paths

Datapath design is commonly used in applications that contain extensive data manipulation, such as 3-D, multimedia, and digital signal processing (DSP). 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 QOR by utilizing the carry-save arithmetic technique.

Beginning with version W-2004.12, Design Compiler provides improved datapath generators and better arithmetic components for both DC Expert and DC Ultra. To take advantage of these enhancements, make sure that the `dw_foundation.sldb` library is listed in the synthetic library and the `synlib_enable_dpgen` variable is set to true (the default). If necessary, use the following commands:

- `set synthetic_library dw_foundation.sldb`
- `set synlib_enable_dpgen true`

These enhancements require a DesignWare license.

Using DC Ultra Datapath Optimization

DC Ultra enables datapath extraction and explores various datapath and resource-sharing options during compile. DC Ultra datapath optimization provides the following benefits:

- Shares datapath operators
- Extracts the 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

DC Ultra datapath optimization is enabled by default. To disable DC Ultra datapath optimization, set `hlo_disable_datapath_optimization` to `true`. (The default is `false`.)

To use the improved datapath generators and better arithmetic components (starting with Design Compiler version W-2004.12), ensure the following settings:

- `set synthetic_library dw_foundation.sldb`
- `set synlib_enable_dpgen true` (default is `true`)

These enhancements require a DesignWare license.

Note:

If you do not specify the `-no_auto_dwlib` option in the `set_ultra_optimization` command and `hlo_disable_datapath_optimization` is `false` (the default), the `dw_foundation.sldb` library is automatically added to the synthetic library list if it is not already there.

DC Ultra datapath optimization requires a DC-Ultra-Features license and a DesignWare-Foundation license.

This section contains the following:

- [Datapath Extraction](#)
- [Two Different Datapath Optimization Methods](#)

- [Methodology Flow](#)
- [Datapath Report](#)
- [Commands Specific to DC Ultra Datapath Optimization](#)

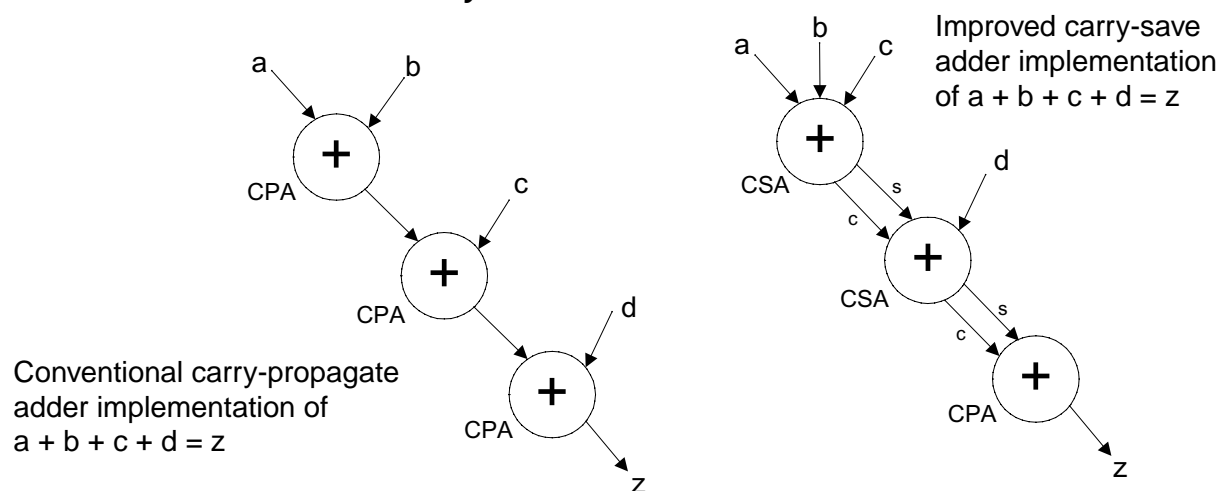
Datapath Extraction

Datapath design is commonly used in applications that contain extensive data manipulation, such as 3-D, multimedia, and digital signal processing (DSP). 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. For example, a 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), as shown in [Figure 8-12](#).

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.

Figure 8-12 *Conventional Carry-Propagate Adder and Faster, Smaller Carry-Save Adder*



DC Ultra datapath optimization can extract the following components:

- Chains of arithmetic operations
- Operators extracted as part of a datapath: $*$, $+$, $-$, $>$, $<$, $<=$, $>=$

Note that comparators are extracted as a part of a datapath.

- MUXs
- Operations with bit truncation
- Shift operators (limited to shifts of constant amounts)

The datapath flow can extract these components only if they are

- Directly connected to each other—that is, no nonarithmetic logic between components
- Not connected to an input or output port

The following components cannot be extracted by any Synopsys datapath methodology:

- Equality and non-equality comparators (==, !=)
- Shift operators of nonconstant amounts
- Operations that have user-specified implementations

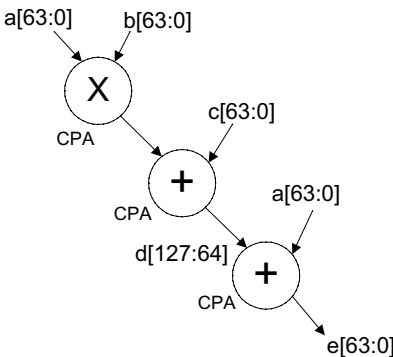
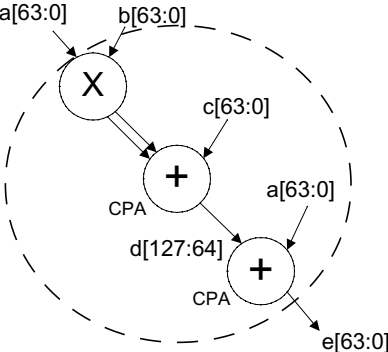
Using DC Ultra datapath optimization, a datapath block can now include truncated bits. This feature is illustrated in [Example 8-7](#).

Example 8-7 Design dp Truncates d After Addition of c

```
module dp (a,b,c,e);  
  
    input [63:0] a,b,c;  
    output [63:0] e;  
    reg [63:0] e;  
    reg [127:0] d;  
  
    always @ (a or b or c)  
    begin  
        d = a * b + c;  
        e = a + d[127:64];  
    end  
  
endmodule
```


In this example, d is truncated after the addition of c. With DC Ultra datapath optimization, operators that follow bit truncation are extracted. [Table 8-9](#) illustrates the greater datapath extraction capabilities of DC Ultra datapath optimization. The dashed line circles those operators that are extracted.

Table 8-9 *DC Ultra Datapath Extraction Supports Bit Truncation*

DC Expert	DC Ultra datapath optimization
 <p>Using DC Expert, no datapath block is extracted.</p> <p>QOR timing: 21.4 area : 304746.42</p>	 <p>Using DC Ultra datapath optimization, the extracted datapath includes the second adder.</p> <p>QOR timing: 4.46 area: 274332.44</p>

When DC Ultra datapath optimization is used to compile design dp in [Example 8-7 on page 8-62](#), the following improvements are realized:

- 350 percent timing improvement, compared to DC Expert results for [Example 8-7](#)

- 10 percent area improvement, compared to DC Expert results for [Example 8-7](#)

Two Different Datapath Optimization Methods

To understand and contrast the two methods for handling datapaths, consider the datapath example defined by the code in [Example 8-8](#).

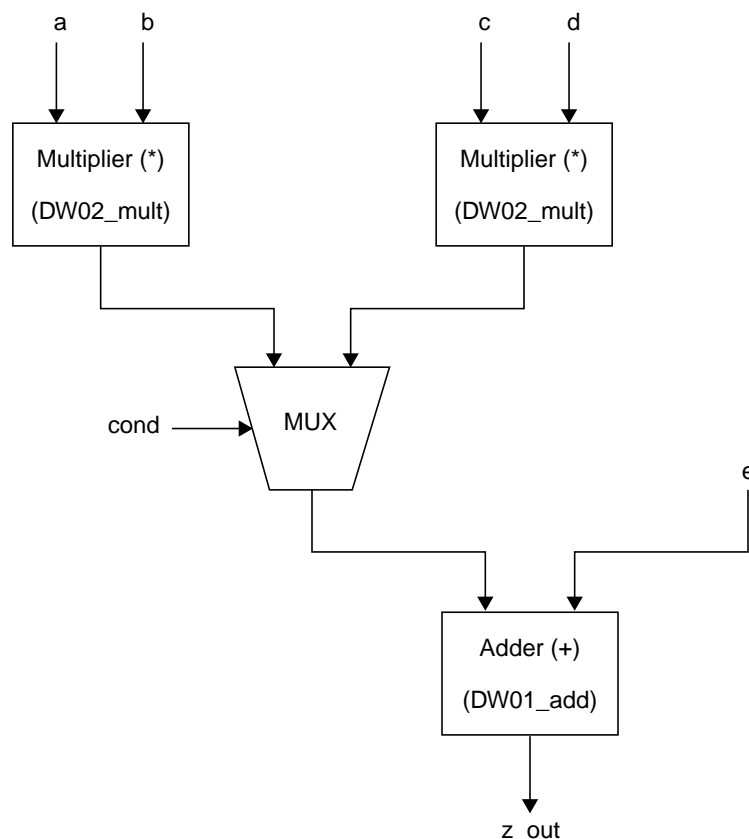
Example 8-8 Design datapath1

```
module datapath1 (clk, cond, a, b, c, d, e, z_out);
    input clk;
    input cond;
    input [3:0] a, b, c, d;
    input [7:0] e;
    output [7:0] z_out;
    reg [7:0] z, mult, z_out;
    always @(*)
        if (cond)
            begin
                mult = a * b;
            end
        else
            begin
                mult = c * d;
            end
    always @(posedge clk)
        begin
            z_out <= mult + e;
        end
endmodule
```

DC Expert Compile

When you compile using DC Expert, the design is optimized into the structure shown in [Figure 8-13](#). Notice that the multipliers and adder in the design are mapped separately to DesignWare parts. It would be advantageous if they were merged into one datapath block, but this is not possible with DC Expert.

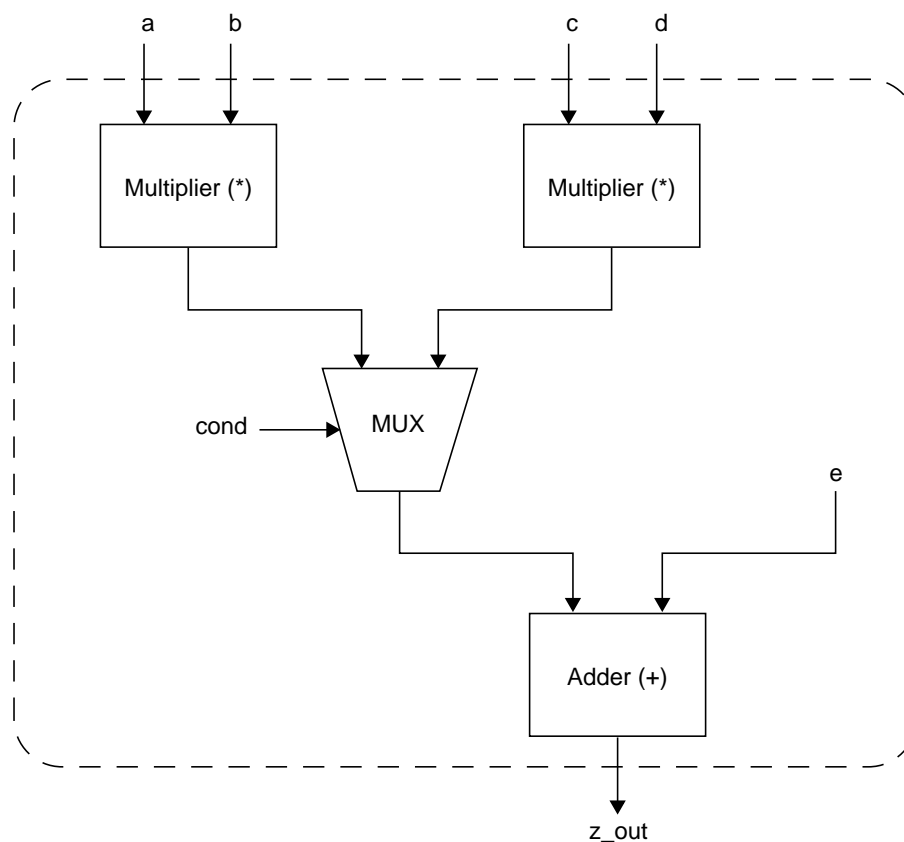
Figure 8-13 DC Expert Optimization



DC Ultra Datapath Optimization

If you compile using DC Ultra datapath optimization, the design is optimized to the structure shown in [Figure 8-14](#). Notice the dashed line around the multipliers, adder, and MUX; this is meant to indicate that DC Ultra datapath optimization has created a single datapath block that includes the arithmetic components.

Figure 8-14 *DC Ultra Datapath Optimization With Datapath Extraction*



Comparison of the Two Datapath Optimizations

In many cases, creating datapath blocks with shared arithmetic components produces better timing, CPU time, area, and QOR.

[Table 8-10](#) gives a comparison summary of the results for the design defined by [Example 8-8 on page 8-64](#).

Table 8-10 *Summary of DC Ultra Datapath Optimization Improvements*

Item	DC Expert	DC Ultra datapath optimization
Critical path slack (report_timing)	-.30	0.11

Table 8-10 Summary of DC Ultra Datapath Optimization Improvements

Item	DC Expert	DC Ultra datapath optimization
Design area (report_area)	3236	2007
Overall compile time	56.6	17.68 Datapath blocks are created during compile.
CPU processing time (CPU time)	143	48

Methodology Flow

To use DC Ultra datapath optimization, follow the guidelines in this section.

Important:

DC Ultra datapath optimization requires both the DC-Ultra-Features license and the DesignWare-Foundation license. It is the only feature in DC Ultra to require both licenses.

1. Set `set_ultra_optimization` to true. This checks out the DC-Ultra-Features license.

When you compile, the following information is displayed in the compile log:

Information: Datapath optimization is enabled. (DP-1)

2. To use the improved datapath generators and better arithmetic components (provided with Design Compiler version W2004.12), ensure the following settings:
 - `set synthetic_library dw_foundation.sldb`

- `set synlib_enable_dpgen true` (default is true)

Note:

To disable DC Ultra datapath optimization, set `hlo_disable_datapath_optimization` to true. The default is false. If set to true, the following message is displayed:

Warning: The built-in datapath optimization in the compile command will be disabled. (DP-3)

3. Use the `report_timing` command to check timing and optimization results.
4. Use `report_resources` to determine which operators were absorbed into the datapath block. In the resource report shown in [Example 8-9](#), the operators absorbed into the datapath are identified.

Example 8-9 *report_resources Using DC Ultra Datapath Optimization for datapath1 Design*
(Example 8-8 on page 8-64)

report_resources

Report : resources

Design : datapath1

Version: W-2004.12

Date : Fri Dec 3 17:15:46 2004

Resource Sharing Report for design datapath1 in file
 /usr/home/..../datapath1.v

Resource	Module	Parameters	Contained Resources	Contained Operations
r262	mult_15_DP_OP_246_3372			mult_11
r264	mult_15_DP_OP_246_3372			mult_15
r266	mult_15_DP_OP_246_3372			add_19

Datapath Report for design datapath1 in file /usr/home/..../datapath1.v

RTL-datapath Connections for mult_15_DP_OP_246_3372-str

RTL Wire	Datapath Port	Direction	Bus Width
c	I1	input	4
d	I2	input	4
a	I3	input	4
b	I4	input	4
e	I5	input	8
N0	C0	control	1
N1	C1	control	1
N26-N19	O1	output	8

Datapath Blocks in mult_15_DP_OP_246_3372-str

Port	Out Width	Datapath Block	Contained Operation_Line	Operation Type
------	-----------	----------------	--------------------------	----------------

Fanout_2	8	I1 * I2	mult_15	UNSIGNED
Fanout_4	8	{ C0 , C1 } ? Fanout_3 : Fanout_2		
			op4	MUX_OP
			op5	MUX_OP
Fanout_3	8	I3 * I4	mult_11	UNSIGNED
O1	8	Fanout_4 + I5	add_19	UNSIGNED

Implementation Report

Cell	Module	Current Implementation	Set Implementation
mult_15_DP_OP_246_3372_1	mult_15_DP_OP_246_3372	str	

No multiplexors to report

1

Notice in this example that when components are absorbed into a datapath block, the resources report (`report_resources` command) also includes a datapath report.

Datapath Report

To further understand how to read the datapath report contained in the resources report, consider the code in [Example 8-10](#).

Example 8-10 Design add: Code

```

1 module add (a,b,c,d,z);
2   input [7:0] a,b,c,d;
3   output [15:0] z;
4   assign z = a + b - c + d;
5 endmodule

```

When this code is compiled using DC Ultra datapath optimization, the `report_resources` command generates the report shown in [Example 8-11](#).

Example 8-11 Datapath Report for Design add

Report : resources

Design : add

Version: W-2004.12

Date : Fri Dec 3 13:44:07 2004

Resource Sharing Report for design add in file

/usr/home/....dp_add.v

Resource	Module	Parameters	Contained Resources	Contained Operations
r256	add_4_2_DP_OP_245_8218			add_1_root_sub_4
r258	add_4_2_DP_OP_245_8218			sub_0_root_sub_4
r260	add_4_2_DP_OP_245_8218			add_4_2

Datapath Report for design add in file /usr/home/....dp_add.v

RTL-datapath Connections for add_4_2_DP_OP_245_8218-str

RTL Wire	Datapath Port	Direction	Bus Width
a	I1	input	8
b	I2	input	8
c	I3	input	8
d	I4	input	8
z	O1	output	16

Datapath Blocks in add_4_2_DP_OP_245_8218-str

Port	Out Width	Datapath Block	Contained Operation_Line	Operation Type
O1	16	I1 + I2 - I3 + I4	add_4_2	UNSIGNED
			sub_0_root_sub_4	UNSIGNED
			add_1_root_sub_4	UNSIGNED
				UNSIGNED

Implementation Report

Cell	Module	Current Implementation	Set Implementation
add_4_2_DP_OP_245_8218_2	add_4_2_DP_OP_245_8218	str	

No multiplexors to report
1

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

- Resource Sharing Report
- Datapath Report
- Implementation Report

From the Resource Sharing Report, you see that there are three arithmetic operators identified as `add_1_root_sub_4`, `sub_0_root_sub_4`, and `add_4_2`. Note that the suffix of the operation names `xxxx_4` represents the line number in the RTL code ([Example 8-10 on page 8-70](#)), and if two adders appear in one line, as in line 4 of the example, the second adder is identified in the report as `xxx_4_2`.

From the Datapath Report you see that the operators are merged into the single datapath module `add_4_2_DP_OP_245_8218-str`. The RTL-datapath Connections table shows the input and output ports of the datapath and their connections to the actual RTL. The Datapath Blocks table shows the datapath expression and operation type.

If you set the `compile_report_dp` variable to true, the Datapath Report is printed to the screen and log file during compile.

Commands Specific to DC Ultra Datapath Optimization

Commands specific to DC Ultra datapath optimization are described in [Table 8-11](#).

Table 8-11 DC Ultra Datapath Optimization Commands

Command	Description
<code>set_ultra_optimization</code>	<p>Set to true to enable DC Ultra; the datapath optimization feature is one of the DC Ultra optimization techniques. Also, set the <code>synlib_enable_dpgen</code> variable to true to utilize optimal datapath generation.</p> <p>If you do not specify the <code>-no_auto_dwlib</code> option, the <code>dw_foundation.sldb</code> library is automatically added to the synthetic library list unless it is already listed. If you do specify this option, the <code>dw_foundation.sldb</code> library is not used.</p> <p>When DC Ultra datapath optimization is enabled, your compile log displays the following message:</p> <p>Information: Datapath optimization is enabled. (DP-1)</p>
<code>hlo_disable_datapath_optimization</code>	<p>Set to true to disable only the DC Ultra datapath optimization feature of DC Ultra. The default is false.</p> <p>Note: DC Ultra datapath optimization requires both the DC-Ultra-Features license and the DesignWare-Foundation license. The foundation license is pulled when you compile. To run DC Ultra if you do not have the foundation license, you must set <code>hlo_disable_datapath_optimization</code> to true; otherwise an error is returned.</p>

For additional information on these commands, see the man pages.

Table 8-12 summarizes the conditions that enable and disable DC Ultra datapath optimization.

Table 8-12 Conditions That Enable and Disable Datapath Optimization

<code>set_ultra_optimization</code>	<code>hlo_disable_datapath_optimization</code>	Datapath optimization enabled (yes or no)
true	false	yes
true	true	no
false	false	no
false	true	no

9

Analyzing and Resolving Design Problems

Use the reports generated by Design Compiler to analyze and debug your design. You can generate reports both before and after you compile your design. Generate reports before compiling to check that you have set attributes, constraints, and design rules properly. Generate reports after compiling to analyze the results and debug your design.

This chapter contains the following sections:

- [Checking for Design Consistency](#)
- [Analyzing Your Design During Optimization](#)
- [Analyzing Design Problems](#)
- [Analyzing Timing Problems](#)
- [Resolving Specific Problems](#)

Checking for Design Consistency

A design is consistent when it does not contain errors such as unconnected ports, constant-valued ports, cells with no input or output pins, mismatches between a cell and its reference, multiple driver nets, connection class violations, or recursive hierarchy definitions.

Use the `check_design` command to verify design consistency. The `check_design` command reports a list of warning and error messages.

- It reports an error if it finds a problem that Design Compiler cannot resolve. You cannot compile a design that has `check_design` errors.

The `check_design` command always reports error messages.

- It reports a warning if it finds a problem that indicates a corrupted design or a design mistake not severe enough to cause the `compile` command to fail.

By default, the `check_design` command reports all warning messages. You can reduce the output by summarizing the warnings (by using the `-summary` option) or by disabling the warnings (by using the `-no_warnings` option).

By default, the `check_design` command validates the entire design hierarchy. To limit the validation to the current design, specify the `-one-level` option.

Analyzing Your Design During Optimization

Design Compiler provides the following capabilities for analyzing your design during optimization:

- It lets you customize the compile log.
- It lets you save intermediate design databases.

The following sections describe these capabilities.

Customizing the 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 four tasks in which Design Compiler works to reduce the compile cost function:

- Delay optimization
- Design rule fixing, phase 1
- Design rule fixing, phase 2
- Area optimization

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.

By default, Design Compiler logs each action in the trials phase by providing the following information:

- Elapsed time
- Design area

- Worst negative slack
- Total negative slack
- Design rule cost
- Endpoint being worked on

You can customize the trials phase output by setting the `compile_log_format` variable. [Table 9-1](#) lists the available data items and the keywords used to select them. For more information about customizing the compile log, see the man page for the `compile_log_format` variable.

Table 9-1 *Compile Log Format Keywords*

Column	Column header	Keyword	Column description
Area	AREA	area	Shows the area of the design.
CPU seconds	CPU SEC	cpu	Shows the process CPU time used (in seconds).
Design rule cost	DESIGN RULE COST	drc	Measures the difference between the actual results and user-specified design rule constraints.
Elapsed time	ELAPSED TIME	elap_time	Tracks the elapsed time since the beginning of the current compile or reoptimization of the design.
Endpoint	ENDPOINT	endpoint	Shows the endpoint being worked on. When delay violations are being fixed, the endpoint is a cell or a port. When design rule violations are being fixed, the endpoint is a net. When area violations are being fixed, no endpoint is printed.

Table 9-1 *Compile Log Format Keywords (Continued)*

Column	Column header	Keyword	Column description
Maximum delay cost	MAX DELAY COST	max_delay	Shows the maximum delay cost of the design.
Megabytes of memory	MBYTES	mem	Shows the process memory used (in MB).
Minimum delay cost	MIN DELAY COST	min_delay	Shows the minimum delay cost of the design.
Path group	PATH GROUP	group_path	Shows the path group of an endpoint.
Time of day	TIME OF DAY	time	Shows the current time.
Total negative slack	TOTAL NEG SLACK	tns	Shows the total negative slack of the design.
Trials	TRIALS	trials	Tracks the number of transformations that the optimizer tried before making the current selection.
Worst negative slack	WORST NEG SLACK	wns	Shows the worst negative slack of the current path group.

Saving Intermediate Design Databases

Design Compiler provides the capability to output an intermediate design database during the trials phase of the optimization process. This capability is called checkpointing. Checkpointing saves the entire hierarchy of the intermediate design. You can use this intermediate design to debug design problems, as described in [“Analyzing Design Problems” on page 9-8](#).

Design Compiler supports both manual checkpointing and automatic checkpointing. The following sections describe these options.

Manual Checkpointing

You can manually checkpoint the design at any time after optimization has entered the trials phase by using the Control-c interrupt. You can checkpoint the design multiple times throughout the optimization process; however, each checkpoint overwrites the previous checkpoint file.

When you are running Design Compiler interactively and press Control-c once, the following menu appears (after a short delay):

```
Please type in one of the following options:
  1 to Write out the current state of the design
  2 to Abort optimization
  3 to Kill the process
  4 to Continue optimization
Please enter a number:
```

Select option 1 to checkpoint the design. By default, Design Compiler writes the intermediate design database to `./CHECKPOINT.db`. You can specify the file name by using the `compile_checkpoint_filename` variable. The directory you specify must exist and be writable. After you checkpoint the design, Design Compiler displays the menu again. Select option 2, 3, or 4, depending on how you want to proceed.

When Design Compiler is running in the background, the behavior of the Control-c interrupt depends on the optimization phase and the number of times you press Control-c.

Before the trials phase,

- Pressing Control-c once stops the optimization process
- Pressing Control-c three times stops the `dc_shell` process

After the trials phase,

- Pressing Control-c once checkpoints the design after a minimum delay of five seconds
- Pressing Control-c three times stops the optimization process
- Pressing Control-c five times stops the dc_shell process

If you use the UNIX `tee` command to print the results of a background run to both the screen and a log file, you must specify the `-i` option of the `tee` command to use checkpointing. For example,

```
% dc_shell -f run.scr | tee -i run.out
```

If you do not use the `-i` option, pressing Control-c interrupts the `tee` process, and the `dc_shell` process never sees the interrupt signal. The `-i` option forces the `tee` process to ignore interrupt signals.

Automatic Checkpointing

You can automatically checkpoint the design based on CPU time intervals, optimization phase, or both.

To checkpoint based on elapsed CPU time, set the `compile_checkpoint_cpu_interval` variable to the required time interval (in minutes). Each checkpoint overwrites the previous checkpoint file.

To checkpoint based on optimization phase, set the `compile_checkpoint_phases` variable to true. This creates a checkpoint file at the following points: before starting delay optimization (pre-delay), before starting the first phase of design rule fixing (pre-DRC1), before starting the second phase of design rule

fixing (pre-DRC2), and before starting area optimization (pre-area). Design Compiler saves each checkpoint in a separate file. [Table 9-2](#) lists the default file name for each phase and the variable used to control each file name. You can turn off checkpointing for any phase by setting the corresponding variable to `none`.

Table 9-2 *Phase-Based Checkpoint Files*

Phase	Default file name	Variable
Pre-delay	./CHECKPOINT_PRE_DELAY.db	compile_checkpoint_pre_delay_filename
Pre-DRC1	./CHECKPOINT_PRE_DRC1.db	compile_checkpoint_pre_drc1_filename
Pre-DRC2	./CHECKPOINT_PRE_DRC2.db	compile_checkpoint_pre_drc2_filename
Pre-area	./CHECKPOINT_PRE_AREA.db	compile_checkpoint_pre_area_filename

Analyzing Design Problems

[Table 9-3](#) shows the design analysis commands provided by Design Compiler. For additional information about these commands, see the man pages.

Table 9-3 *Commands to Analyze Design Objects*

Object	Command	Description
Design	report_design report_area report_hierarchy report_resources	Reports design characteristics. Reports design size and object counts. Reports design hierarchy. Reports resource implementations.
Instances	report_cell	Displays information about instances.
References	report_reference	Displays information about references.
Pins	report_transitive_fanin report_transitive_fanout	Reports fanin logic. Reports fanout logic.

Table 9-3 Commands to Analyze Design Objects (Continued)

Object	Command	Description
Ports	<code>report_port</code>	Displays information about ports.
	<code>report_bus</code>	Displays information about bused ports.
	<code>report_transitive_fanin</code>	Reports fanin logic.
	<code>report_transitive_fanout</code>	Reports fanout logic.
Nets	<code>report_net</code>	Reports net characteristics.
	<code>report_bus</code>	Reports bused net characteristics.
	<code>report_transitive_fanin</code>	Reports fanin logic.
	<code>report_transitive_fanout</code>	Reports fanout logic.
Clocks	<code>report_clock</code>	Displays information about clocks.

Analyzing Timing Problems

Before you begin debugging timing problems, verify that your design meets the following requirements:

- You have defined the operating conditions.
- You have specified realistic constraints.
- You have appropriately budgeted the timing constraints.
- You have properly constrained the paths.
- You have described the clock skew.

If your design does not meet these requirements, make sure it does before you proceed.

After producing the initial mapped netlist, use the `report_constraint` command to check your design's performance.

Table 9-4 lists the timing analysis commands.

Table 9-4 Timing Analysis Commands

Command	Analysis task description
<code>report_design</code>	Shows operating conditions, wire load model and mode, timing ranges, internal input and output, and disabled timing arcs.
<code>check_timing</code>	Checks for unconstrained timing paths and clock-gating logic.
<code>report_port</code>	Shows unconstrained input and output ports and port loading.
<code>report_timing_requirements</code>	Shows all timing exceptions set on the design.
<code>report_clock</code>	Checks the clock definition and clock skew information.
<code>derive_clocks</code>	Checks internal clock and unused registers.
<code>report_path_group</code>	Shows all timing path groups in the design.
<code>report_timing</code>	Checks the timing of the design.
<code>report_constraint</code>	Checks the design constraints.
<code>report_delay_calculation</code>	Reports the details of a delay arc calculation.

Resolving Specific Problems

This section provides examples of design problems you might encounter and describes the workarounds for them.

Analyzing Cell Delays

Some cell delays shown in the full path timing report might seem too large. Use the `report_delay_calculation` command to determine how Design Compiler calculated a particular delay value.

[Example 9-1](#) shows a full path timing report with a large cell delay value.

Example 9-1 Full Path Timing Report

```
*****
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : Adder8
Version: 1999.05
Date   : Mon Jan 4 10:56:49 1999
*****
```

Operating Conditions:
Wire Loading Model Mode: top

Startpoint: cin (input port)
Endpoint: cout (output port)
Path Group: (none)
Path Type: max

Point	Incr	Path
input external delay	0.00	0.00 f
cin (in)	0.00	0.00 f
U19/Z (AN2)	0.87	0.87 f
U18/Z (EO)	1.13	2.00 f
add_8/U1_1/CO (FA1A)	2.27	4.27 f
add_8/U1_2/CO (FA1A)	1.17	5.45 f
add_8/U1_3/CO (FA1A)	1.17	6.62 f
add_8/U1_4/CO (FA1A)	1.17	7.80 f
add_8/U1_5/CO (FA1A)	1.17	8.97 f
add_8/U1_6/CO (FA1A)	1.17	10.14 f
add_8/U1_7/CO (FA1A)	1.17	11.32 f
U2/Z (EO)	1.06	12.38 f
cout (out)	0.00	12.38 f
data arrival time		12.38 f

(Path is unconstrained)

The delay from port cin through cell FA1A seems large (2.27 ns). Enter the following command to determine how Design Compiler calculated this delay:

```
dc_shell> report_delay_calculation \  
          -from add_8/U1_1/A -to add_8/U1_1/CO
```


Example 9-2 shows the results of this command.

Example 9-2 Delay Calculation Report

```
*****
Report : delay_calculation
Design : Adder8
Version: 1997.01
Date   : Mon Apr  7 13:23:12 1997
*****

From pin:                add_8/U1_1/A
To pin:                  add_8/U1_1/CO

arc sense:               unate
arc type:                cell
Input net transition times: Dt_rise = 0.1458, Dt_fall = 0.0653

Rise Delay computation:
rise_intrinsic           1.89 +
rise_slope * Dt_rise      0 * 0.1458 +
rise_resistance * (pin_cap + wire_cap) / driver_count
0.1458 * (2 + 0) / 1
-----
Total                    2.1816

Fall Delay computation:
fall_intrinsic           2.14 +
fall_slope * Dt_fall      0 * 0.0653 +
fall_resistance * (pin_cap + wire_cap) / driver_count
0.0669 * (2 + 0) / 1
-----
Total                    2.2738
```

Finding Unmapped Cells

All unmapped cells have the `is_unmapped` attribute. You can use the `dcsh filter find` and `find` commands or the `dctcl get_cells` command to locate all unmapped components:

```
dc_shell> filter find(-hier, cell, "**") "@is_unmapped==true"
```

```
dc_shell-t> get_cells -hier -filter "@is_unmapped==true"
```

Finding Black Box Cells

All black box cells have the `is_black_box` attribute. You can use the `dcsh filter` and `find` commands or the `dctl get_cells` command to locate all black box cells:

```
dc_shell> filter find(-hier, cell, "**") \  
           "@is_black_box==true"
```

```
dc_shell-t> get_cells -hier -filter "@is_black_box==true"
```

Finding Hierarchical Cells

All hierarchical cells have the `is_hierarchical` attribute. You can use the `dcsh filter` and `find` commands or the `Tcl get_designs` command to locate all hierarchical cells:

```
dc_shell> filter find(design, "**") "@is_hierarchical==true"
```

```
dc_shell-t> get_designs -filter "@is_hierarchical==true"
```

Disabling Reporting of Scan Chain Violations

If your design contains scan chains, it is likely that these chains are not designed to run at system speed. This can cause false violation messages when you perform timing analysis. To mask these messages, use the `set_disable_timing` command to break the scan-related timing paths (scan input to scan output and scan enable to scan output).

```
dc_shell> set_disable_timing my_lib/scanf \  
           -from TI -to Q  
dc_shell> set_disable_timing my_lib/scanf \  
           -from CP -to TE
```

This example assumes that

- scanf is the scan cell in your technology library
- TI is the scan input pin on the scanf cell
- TE is the scan enable on the scanf cell
- Q is the scan output pin on the scanf cell

[Example 9-3](#) and [Example 9-4](#) show scripts that you can use to identify the scan pins in your technology library.

Example 9-3 Script to Identify Scan Pins (dcsh Mode)

```
filter find(cell, my_lib/*) "@is_sequential==true"
seq_cell_list = dc_shell_status
foreach (seq_cell, seq_cell_list) {
    seq_pins = seq_cell + "/*"
    filter find(pin, seq_pins) "@signal_type==test_scan_in"
    si = dc_shell_status
    if (si) {
        echo "Scan pins for cell " seq_cell
        echo "    scan input: " si
        filter find(pin, seq_pins) "@signal_type==test_scan_out"
        echo "    scan output: " dc_shell_status
    }
}
```

Example 9-4 Script to Identify Scan Pins (dctcl Mode)

```
set seq_cell_list [get_cells class/* -filter "@is_sequential==true"]
foreach_in_collection seq_cell $seq_cell_list {
    set seq_pins "[get_object_name $seq_cell]/*"
    set si [get_pins $seq_pins -filter "@signal_type==test_scan_in"]
    if {[sizeof_collection $si] > 0} then {
        echo "Scan pins for cell [get_object_name $seq_cell]"
        echo "    scan input: [get_object_name $si]"
        echo "    scan output: [get_object_name [get_pins $seq_pins \
            -filter "@signal_type==test_scan_out"]]"
    }
}
```

Insulating Interblock Loading

Design Compiler determines load distribution in the driving block. If a single output port drives many blocks, a huge incremental cell delay can result. To insulate the interblock loading, fan the heavily loaded net to multiple output ports in the driving block. Evenly divide the total load among these output ports.

Preserving Dangling Logic

By default, Design Compiler optimizes away dangling logic. Use one of the following methods to preserve dangling logic (for example, spare cells) during optimization:

- Place the `dont_touch` attribute on the dangling logic.
- Connect the dangling logic to a dummy port.

Preventing Wire Delays on Ports

If your design contains unwanted wire delays between ports and I/O cells, you can remove these wire delays by specifying zero resistance (infinite drive strength) on the net. Use the `set_resistance` command to specify the net resistance. For example, enter one of the following commands (depending on your shell mode):

```
dc_shell> set_resistance 0 find(net, wire_io4)
```

```
dc_shell-t> set_resistance 0 [get_nets wire_io4]
```

Breaking a Feedback Loop

Follow these steps to break a feedback loop in your design:

1. Find the feedback loop in your design by using the `report_timing -loop` option.
2. Break the feedback loop by specifying the path as a false path.

Analyzing Buffer Problems

Note:

This section uses the term *buffer* to indicate either a buffer or an inverter chain.

This section describes the following topics:

- Buffer insertion behavior
- Missing buffer problems
- Extra buffer problems
- Hanging buffer problems
- Modified buffer network problems

Understanding Buffer Insertion

Design Compiler inserts buffers to correct maximum fanout load or maximum transition time violations. If Design Compiler does not insert buffers during optimization, the tool probably does not identify a violation. For more information about the maximum fanout load and maximum transition time design rules, see [“Setting Design Rule Constraints” on page 7-3](#).

Use the `report_constraint` command to get details on constraint violations.

Figure 9-1 shows a design containing the IV1 cell.

Figure 9-1 Buffering Example

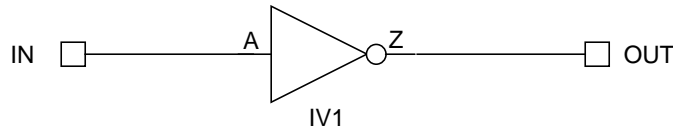


Table 9-5 gives the attributes defined in the technology library for the IV1 cell.

Table 9-5 IV1 Library Attributes

Pin	Attribute	Value
A	direction	input
	capacitance	1.5
	fanout_load	1
Z	direction	output
	rise_resistance	0.75
	fall_resistance	0.75
	max_fanout	3
	max_transition	2.5

Example 9-3 shows the constraint report generated by the command sequences shown in Table 9-6.

Table 9-6 `report_constraint` Examples

dcsh Command Sequence	dctl Command Sequence
<pre> set_drive 0 find(port,IN) set_load 0 find(port,OUT) report_constraint </pre>	<pre> set_drive 0 [get_ports IN] set_load 0 [get_ports OUT] report_constraint </pre>

Example 9-5 Constraint Report

```
*****
Report : constraint
Design : buffer_example
Version: 1999.05
Date   : Mon Jan 4 10:56:49 1999
*****
```

Constraint	Cost
-----	-----
max_transition	0.00 (MET)
max_fanout	0.00 (MET)

To see the constraint cost functions used by Design Compiler, specify the `-verbose` option of the `report_constraint` command (shown in Example 9-4).

Example 9-6 Constraint Report (-verbose)

```
*****
Report : constraint
        -verbose
Design : buffer_example
Version: 1999.05
Date   : Mon Jan 4 10:56:49 1999
*****
```

Net: OUT

max_transition	2.50
- Transition Time	0.00
-----	-----
Slack	2.50 (MET)

Net: OUT

max_fanout	3.00
- Fanout	0.00
-----	-----
Slack	3.00 (MET)

The verbose constraint report shows that two constraints are measured:

- Maximum transition time (2.50)
- Maximum fanout load (3.00)

Design Compiler derives the constraint values from the attribute values on the output pin of the IV1 cell.

When you compile this design, Design Compiler does not modify the design because the design meets the specified constraints.

To list all constraint violations, use the `-all_violators` option of the `report_constraint` command (shown in [Example 9-7](#)).

Example 9-7 Constraint Report (-all_violators)

```
*****
Report : constraint
        -all_violators
Design : buffer_example
Version: 1999.05
Date   : Mon Jan 4 10:56:49 1999
*****
```

This design has no violated constraints.

This design does not have any constraint violations. Changing the port attributes, however, can cause constraint violations to occur. Example 9-6 shows the result of the following command sequence:

```
dc_shell> set_drive 2.5 IN
dc_shell> set_max_fanout 0.75 IN
dc_shell> set_load 4 OUT
dc_shell> set_fanout_load 3.5 OUT
dc_shell> report_constraint -all_violators -verbose
```


Example 9-8 Constraint Report (After Port Attributes Are Modified)

```
*****
Report : constraint
        -all_violators
        -verbose
Design : buffer_example
Version: 1999.05
Date   : Mon Jan 4 10:56:49 1999
*****

Net: OUT

max_transition      2.50
- Transition Time   3.00
-----
Slack               -0.50 (VIOLATED)

Net: OUT

max_fanout          3.00
- Fanout            3.50
-----
Slack               -0.50 (VIOLATED)

Net: IN

max_fanout          0.75
- Fanout            1.00
-----
Slack               -0.25 (VIOLATED)
```

This design now contains three violations:

- Maximum transition time violation at OUT
Actual transition time is $4.00 * 0.75 = 3.00$, which is greater than the maximum transition time of 2.50.
- Maximum fanout load violation at OUT
Actual fanout load is 3.5, which is greater than the maximum fanout load of 3.00.

- Maximum fanout load violation at IN

Actual fanout load is 1.00, which is greater than the maximum fanout load of 0.75.

There is no `max_transition` violation at IN, even though the transition time on this net is $2.5 * 1.5 = 3.75$, which is well above the `max_transition` requirement of 2.50. Design Compiler does not recognize this as a violation because the requirement of 2.50 is a design rule from the output pin of cell IV1. This requirement applies only to a net driven by this pin. The IV1 output pin does not drive the net connected to port IN, so the `max_transition` constraint does not apply to this net.

If you want to constrain the net attached to port IN to a maximum transition time of 2.50, enter one of the following commands (depending on your shell mode):

```
dc_shell> set_max_transition 2.5 find(port,IN)
```

```
dc_shell-t> set_max_transition 2.5 [get_ports IN]
```

This command causes `report_constraint -verbose -all_violators` to add the following lines to the report shown in Example 9-6:

```
Net: IN
max_transition          2.50
- Transition Time      3.75
-----
Slack                  -1.25  (VIOLATED)
```

When you compile this design, Design Compiler adds buffering to correct the `max_transition` violations.

Remember the following points when you work with buffers in Design Compiler:

- The `max_fanout` and `max_transition` constraints control buffering; be sure you understand how each is used.
- Design Compiler fixes only violations it detects.
- The `report_constraint` command identifies any violations.

Correcting for Missing Buffers

Missing buffers present the most frequent buffering problem. It usually results from one of the following conditions:

- Incorrectly specified constraints
- Improperly constrained designs
- Incorrect assumptions about constraint behavior

To debug the problem, generate a constraint report (`report_constraint`) to determine whether Design Compiler recognized any violations.

If Design Compiler reports no `max_fanout` or `max_transition` violations, check the following:

- Are constraints applied?
- Is the library modeled for the correct attributes?
- Are the constraints tight enough?

If Design Compiler recognizes a violation but `compile` does not insert buffers to remove the violation, check the following:

- Does the violation exist after compile?

- Are there `dont_touch` or `dont_touch_network` attributes?
- Are there three-state pins that require buffering?
- Have you considered that `max_transition` takes precedence over `max_fanout`?

Incorrectly Specified Constraints. A vendor might omit an attribute you want to use, such as `fanout_load`. If a vendor has not set this attribute in the library, Design Compiler does not find any violations for the constraint. You can check whether attributes have been assigned to cell pins by using the `get_attribute` command with the `dcsh find` or the `Tcl get_pins` command. For example, to determine whether a pin has a `fanout_load` attribute, enter

```
dc_shell> get_attribute \
          find(pin,library/cell/pin) fanout_load
```

```
dc_shell-t> get_attribute \
            [get_pins library/cell/pin] fanout_load
```

The vendor might have defined `default_fanout_load` in the library. If this value is set to zero or to an extremely small number, any pin that does not have an explicit `fanout_load` attribute inherits this value.

Improperly Constrained Designs. Occasionally, a vendor uses extremely small capacitance values (on the order of 0.001). If your scripts do not take this into account, you might not be constraining your design tightly enough. Try setting an extreme value, such as 0.00001, and run `report_constraint` to make sure a violation occurs.

You can use the `load_of` command with the `dcsh find` or the `Tcl get_pins` command to check the capacitance values in the technology library:

```
dc_shell> load_of find(pin,library/cell/pin)
```

```
dc_shell-t> load_of [get_pins library/cell/pin]
```

Incorrect Assumptions About Constraint Behavior. Check to make sure you are not overlooking one of the following aspects of constraint behavior:

- A common mistake is the assumption that the `default_max_transition` or the `default_max_fanout` constraint in the technology library applies to input ports. These constraints apply only to the output pins of cells within the library.
- Maximum transition time takes precedence over maximum fanout load within Design Compiler. Therefore, a maximum fanout violation might not be corrected if the correction affects the maximum transition time of a net.
- Design Compiler might have removed a violation by sizing gates or modifying the structure of the design.

Generate a constraint report after optimization to verify that the violation still exists.

- Design Compiler cannot correct violations if `dont_touch` attributes exist on the violating path.

You might have inadvertently placed `dont_touch` attributes on a design or cell reference within the hierarchy. If so, Design Compiler reports violations but cannot correct them during optimization.

Use the `report_cell` command and the `get_attribute` command to see whether these attributes exist.

- Design Compiler cannot correct violations if `dont_touch_network` attributes exist on the violating path.

If you have set the `dont_touch_network` attribute on a port or pin in the design, all elements in the transitive fanout of that port or pin inherit the attribute. If this attribute is set, Design Compiler reports violations but does not modify the network during optimization.

Use the `remove_attribute` command to remove this attribute from the port or net.

- Design Compiler does not support additional buffering on three-state pins.

For simple three-state cells, Design Compiler attempts to enlarge a three-state cell to a stronger three-state cell.

For complex three-state cells, such as sequential elements or RAM cells, Design Compiler cannot build the logic necessary to duplicate the required functionality. In such cases, you must manually add the extra logic or rewrite the source HDL to decrease the fanout load of such nets.

Correcting for Extra Buffers

Extremely conservative numbers for `max_transition`, `max_fanout`, or `max_capacitance` force Design Compiler to buffer nets excessively. If your design has an excessive number of buffers, check the accuracy of the design rule constraints applied to the design.

If you have specified design rule constraints that are more restrictive than those specified in the technology library, evaluate the necessity for these restrictive design rules.

You can debug this type of problem by setting the priority of the maximum delay cost function higher than the maximum design rule cost functions (using the `set_cost_priority -delay` command). Changing the priority prevents Design Compiler from fixing the maximum design rule violations if the fix results in a timing violation.

Correcting for Hanging Buffers

A buffer that does not fan out to any cells is called a hanging buffer. Hanging buffers often occur because the buffer cells have `dont_touch` attributes. These attributes either can be set by you, in the hope of retaining a buffer network, or can be inherited from a library.

The `dont_touch` attribute on a cell signals to Design Compiler that the cell should not be touched during optimization. Design Compiler follows these instructions by leaving the cell in the design. But because the buffer might not be needed to meet the constraints that are set, Design Compiler disconnects the net from the output. The design meets your constraints, but because the cell has the `dont_touch` attribute, the cell cannot be removed. Remove the `dont_touch` attribute to correct this problem.

Correcting Modified Buffer Networks

Sometimes it appears that Design Compiler modifies a buffer network that has `dont_touch` attributes. This problem usually occurs when you place the `dont_touch` attribute on a cell and expect the cells adjacent to that cell to remain in the design.

Design Compiler does not affect the cell itself but modifies the surrounding nets and cells to attain the optimal structure. If you are confident about the structure you want, you can use one of the following strategies to preserve your buffer network:

- Group the cells into a new hierarchy and set `dont_touch` attributes on that hierarchy.
- Set the `dont_touch_network` attribute on the pin that begins the network.
- Set the `dont_touch` attribute on all cells and nets within the network that you want to retain.

10

Creating, Instantiating, and Using Interface Logic Models in Hierarchical Synthesis

An interface logic model (ILM) is a structural model of a circuit that is modeled as a smaller circuit representing the interface logic of the block. The interface logic model contains the cells whose timing is affected by or affects the external environment of a block. Using interface logic models enhances capacity and reduces runtime for top-level optimization.

The concepts and tasks necessary to use interface logic models are described in the following sections:

- [Introduction to Interface Logic Models](#)
- [Benefits of Interface Logic Models](#)
- [Considerations When Using Interface Logic Models](#)
- [Interface Logic Model Handling by Design Compiler](#)

- Creating Interface Logic Models
- Reporting Information About Interface Logic Models
- Instantiating and Using Interface Logic Models in Logical Synthesis
- Back-Annotation With Interface Logic Models
- Summary of Commands for Interface Logic Models

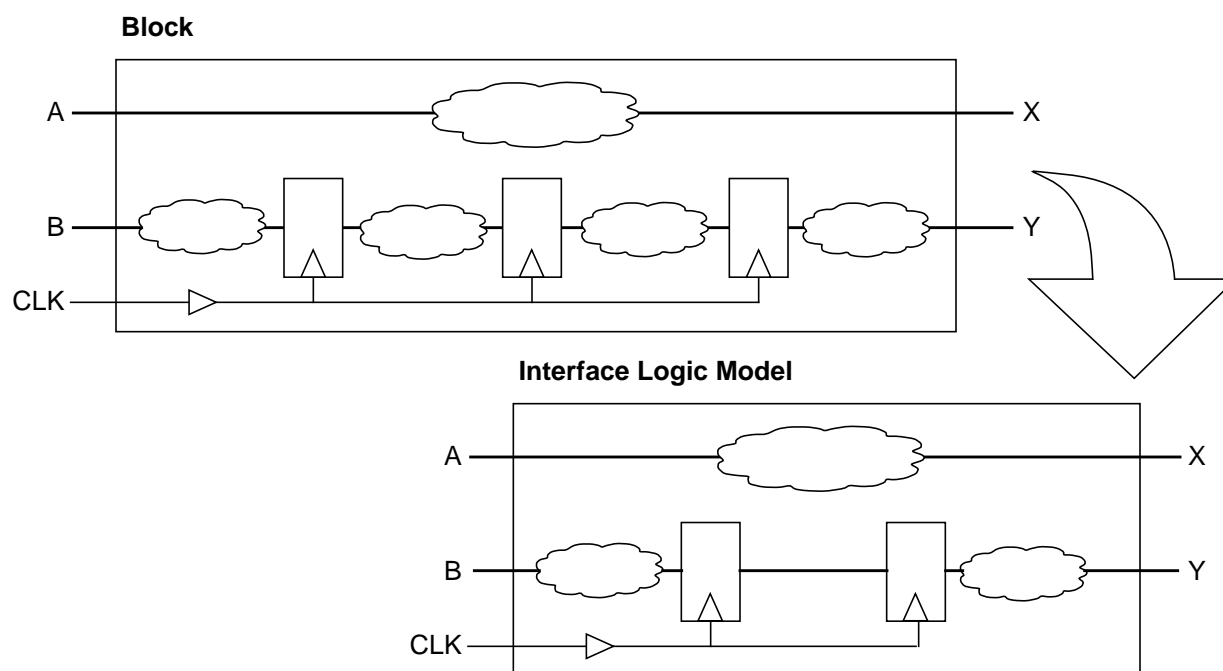
Introduction to Interface Logic Models

When you perform a top-level synthesis of a large hierarchical design, capacity and runtime issues can arise. One way you can address these problems is to represent the compiled subblocks of such a design with interface logic models (ILMs).

The interface logic model of a compiled block contains the mapped interface logic of the block and any logic that affects interface performance. All other logic is removed. That is, the gate-level netlist of the block is replaced by a gate-level netlist that preserves only the interface logic and the logic affecting interface behavior. The resulting model design is usually much smaller than the original block. Thus, replacing the subblocks with their smaller interface logic models can help alleviate capacity problems and lead to improved compile times for the top-level design.

Figure 10-1 illustrates the relationship between a compiled block and its interface logic model representation. The logic from the input port to the first register is preserved, as is the logic from the last register to the output port. Clock connections to the preserved registers are kept as well. Pure combinational logic that is associated with the input port to output port timing paths (A to X) is also preserved.

Figure 10-1 A Block and Its Interface Logic Model



When you create interface logic models for hierarchical scan synthesis, the block's scan chains are preserved if you used a test model. The scan chains are not preserved in the gate-level model. For more information on using test models with ILMs, see the *DFT Compiler Scan Synthesis User Guide*.

Important:

Synthesis interface logic models support only SDF and `set_load` information for back-annotation.

Benefits of Interface Logic Models

In addition to improved capacity and reduced compile times for very large designs that include interface logic models (chip-level synthesis), these models provide the following additional benefits:

- Interface logic models preserve interface logic without modifying it. They do not abstract the logic but instead discard only what is not required for modeling boundary timing. Any block hierarchy involved in the boundary timing is retained.
- An interface logic model is context independent because it is based on the original netlist and contains instantiations of library cells that constitute the interface logic of the design's actual interface.
- Interface logic models provide highly accurate timing representations of the original design interface behavior. For typical designs and technologies, these models preserve the original block timing to within plus or minus 10 picoseconds.
- Interface logic models that Design Compiler generates use the area of the original block netlist to determine which wire load model is selected for the model. This approach contributes significantly to the timing accuracy of the model.
- Any back-annotated data (`set_load` and SDF) that was applied to the original netlist is retained by the interface logic model. Therefore, you can optionally propagate this back-annotated data up to the top level. By using this option, you can avoid timing inaccuracies that would be incurred if the tool had to reestimate delays for nets fully enclosed in the interface logic model.

- Differences in the timing characteristics between an interface logic model and the original netlist are easy to identify because the interface logic is maintained, clock and data paths are preserved without modification, and cell and net names are identical. These factors help you isolate the cause of timing discrepancies.
- The runtime for creating interface logic models is fast because the tool's identification of the design's interface logic is a structural operation achieved by analysis of the circuit topology.
- An interface logic model can be used for an instance nested anywhere in the design hierarchy, including within another interface logic model.
- Interface logic models can be saved as .db files with no loss of attributes related to the interface logic.
- Interface logic models generated by Design Compiler are supported by other Synopsys tools, such as Floorplan Compiler, Physical Compiler, PrimeTime, and DFT Compiler.

Considerations When Using Interface Logic Models

When you use interface logic models, keep these points in mind:

- The amount of improvement for memory and runtime depends on design style.

Some design types, such as pure combinational logic blocks or latch-based designs with many levels of time borrowing, do not show much reduction. The interface logic for such designs tends to contain much of the original design.

Designs with registered inputs and outputs have the greatest reduction in size (original netlist compared with its interface logic model).

- For nested ILMs, you can control what logic from a lower-level ILM is included in the upper-level ILM. Specifically, you can control whether all the lower-level ILM logic or only the lower-level ILM logic involved in the upper-level timing paths is included in the upper-level ILM.
- By using the `-traverse_disabled_arcs` option with the `create_ilm` command, a single ILM can be created for a block that has timing arcs or paths disabled by a timing exception command (for example, `set_false_path`, `set_disable_timing`, and so forth) or by the `set_case_analysis` command. If you do not use this option, you will need to create multiple ILMs, each one valid only for its particular case.
- Timing exceptions specified on a design's core logic (the register-to-register logic that is not part of the interface) are lost in interface logic model representation.
- Timing exceptions related to the interface are saved in the model but are not automatically visible to the top-level design. To propagate interface logic model exceptions to the top level (current design), use the `propagate_constraints` command.
- User-controlled handling of side-load cells (capacitance) can affect the timing of an interface path even when side-load cells are not part of the interface logic of the original design.
- Propagated clocks retain side loads if the appropriate option is set.

- Generated clocks are treated like clock ports. Interface registers driven by a generated clock are retained in the model. However, internal registers driven by generated clocks are not retained.
- By default, latches found in interface logic are assumed to be potential time borrowers; they are treated like combinational logic gates. You can specify the number of latch levels over which time borrowing can occur for latch chains that are a part of the interface logic.
- Designs containing interface logic models can be budgeted. For more information on budgeting with ILMs, see the *Budgeting for Synthesis User Guide*.
- During chip-level optimizations, the tool does not optimize cells within interface logic models.

Interface Logic Model Handling by Design Compiler

Design Compiler handles interface logic models as follows:

- Maintains the hierarchy of the original netlist being modeled as an interface logic model. You can flatten an ILM by ungrouping it, using the `ungroup -all -flatten` command.
- Automatically applies the `dont_touch` attribute to interface logic model cells.
- Allows propagated clocks to retain side loads if the option that instructs Design Compiler to keep side loads is set. Generated clocks are treated like clock ports. Interface registers driven by the generated clock are retained in the model. Registers driven by the generated clock that are in internal register-to-register paths are not included in the model.

Note:

Any block-level subdesign that contains multiple instantiations of a hierarchical cell is automatically uniquified when you run the `compile` command on the subdesign. You can then create an ILM for the block and use that ILM in the top-level design.

A top-level design that contains multiply instantiated ILMs does not need to be uniquified (`uniquify` command) because of the presence of the ILMs. Note, however, that running the `compile` command at the top level does *not* automatically uniquify any multiply instantiated ILMs.

If a block-level subdesign has a `dont_touch` attribute set on it, you should remove the attribute if you want to create a compact ILM. If you do not remove the attribute, the size of the ILM is likely to increase, causing the `create_ilm` command to issue ILM-20 warning messages.

Guidelines for Creating an Interface Logic Model

When you use an interface logic model, observe these guidelines:

- To reduce the model size, use the `-ignore_ports` option or the `-auto_ignore` option (input ports only) with the `create_ilm` command. With this option, the command
 - Ignores ports that are startpoints or endpoints of a false path if the `-traverse_disabled_arcs` option has not been used.
 - Ignores reset and scan enable ports.

Note:

To have the tool automatically fix DRC errors that might result from using the `-ignore_ports` or `-auto_ignore` options, also use the `-keep_boundary_cells` option with the `create_ilm` command. This option retains the boundary cells connected to the ignored ports.

- To include all side-load cells in the ILM, use the `-include_side_load all` option. Using this option increases the model size but can improve model timing accuracy.
- Use the `report_annotated_delay` command to check if the necessary nets and cells are annotated for a postroute SDF flow.
- Apply back-annotation on interface logic models for the postroute flow. See [“Applying Back-Annotation Data for Interface Logic Models” on page 10-34](#).
- Handling “what if” constraints.

If you choose to apply constraints such as `set_case_analysis`, `set_disable_timing`, or `set_false_path` on a block, the logic included in the ILM created for the block might be pruned incorrectly for all applications where the ILM could be used. (Logic pruning occurs when an ILM is created.)

However, when you create an ILM for a block that has disabled timing arcs or pins (for example, from case analysis, disabled timing arcs, false paths, and so on), you can prevent pruning by using the `-traverse_disabled_arcs` option with the `create_ilm` command.

Note that if the `-traverse_disabled_arcs` option is used with the `create_ilm` command, you might need to propagate case analysis constraints from the block level to the top level by

using the `propagate_constraints` command. Not doing so could lead to including additional timing paths that would need to be disabled for timing analysis when the ILMs are employed at the top level.

Alternatively, do not apply such constraints to a block if you are not going to use the `-traverse_disabled_arcs` option with the `create_ilm` command when you create the ILM for the block.

Note that either method creates an ILM that is independent of case analysis. Case analysis can then be applied to the block's ILM as desired for the particular application.

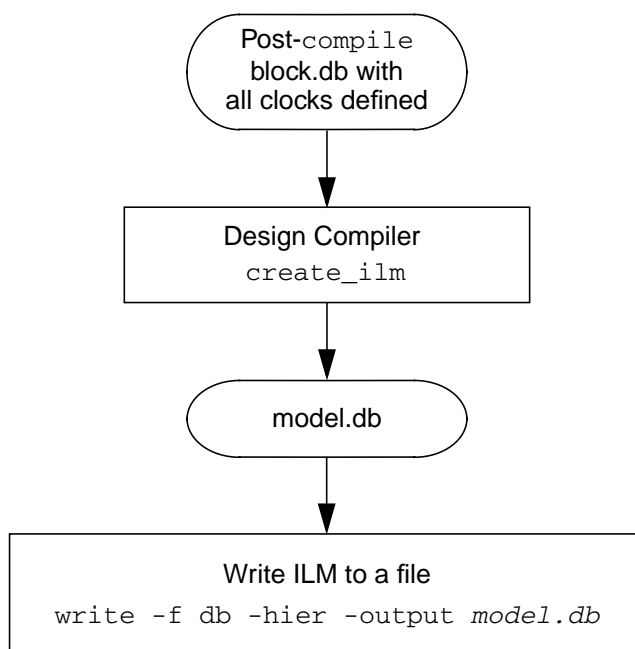
Creating Interface Logic Models

You use the `create_ilm` command to generate an interface logic model. Basically, this command

- Identifies the interface logic to be included in the model. You can specify various command options to control what logic is included as part of a block's interface.
- Extracts the logic and builds the model. Certain command options can be used to include side-load cells (technically not part of the interface logic) and to include physical design information in the model.

Figure 10-2 shows the flow for creating interface logic models.

Figure 10-2 Flow for Creating Interface Logic Models in Design Compiler



This flow follows these major steps:

1. Determine the block(s) you want to model as interface logic models.
2. Read and link the mapped block.db file for the block you are modeling. Use the `read_db` and `link` commands.
3. Identify all clocks by using the appropriate commands, for example, `create_clock` or `create_generated_clock`.
4. Create the interface logic model for each block you chose. Use the `create_ilm` command with appropriate options as needed.

After command execution, the design in memory is the ILM for the block. It has the same name as the original design and is the current design.

Note:

In special situations where you want to create a customized model that selectively includes (or excludes) certain cells, nets, and pins, you can separately use the `create_ilm -identify_only` and `create_ilm -extract_only` commands. Using these options gives you the flexibility to create an ILM for cases where you want to augment what would otherwise be included in the ILM produced by `create_ilm`. You should use the options only in such cases.

5. If you want to save the model to a file, you must use the command `write -f db -hier -output model.db` to write the model to a file. See [“Saving Interface Logic Models” on page 10-21](#).

Note:

If the current design contains nested interface logic models, you can control what logic from a lower-level ILM is included in the upper-level ILM. That is, you can control whether all the lower-level ILM logic or only the lower-level ILM logic involved in the upper-level interface timing paths is included in the upper-level ILM. For more information, see [Example 10-3 on page 10-26](#).

Checking the Interface Logic Model

When you run the `create_ilm` command, the following automatic checks are carried out:

- If the current design contains sequential elements but no clock is defined, the command stops with an error message, and no ILM is created.
- If the current design does not contain any sequential elements, a warning message is issued and the ILM is created. However, this ILM is the same size as the original design.

Controlling the Logic Included in an Interface Logic Model

Interface logic contains the circuitry from the following:

- Leaf cells and macro cells in timing paths that lead from input ports to output ports (combinational input to output paths)
- Leaf cells and macro cells in timing paths that lead from input ports to edge-triggered registers
- Leaf cells and macro cells in timing paths that lead to output ports from edge-triggered registers
- Macros that are not part of the interface timing paths but are included in the model when the `-keep_macros` option of the `create_ilm` command is used

This capability is useful when an ILM is taken into a floor-planning tool for the purpose of exploring potential floorplan changes that involve the ILM. If you do not use this option, macros that are not part of the interface logic of the design are excluded from the ILM.

- Boundary cells of ignored ports that you include by using the `-keep_boundary_cells` option of the `create_ilm` command.

You specify ignored ports by using the `-ignore_ports` or `-auto_ignore` option.

- Clock trees that drive interface registers, including registers in the clock tree
- Clock-gating circuitry, if it is driven by external ports

- Any transparent latch encountered in a timing path from an input port to a register or from an output register to an output port is treated as a combinational logic device and is included in the interface logic. You can control this behavior by using the `-latch_level` option of the `create_ilm` command. (See [Figure 10-3 on page 10-17](#) and [Figure 10-4 on page 10-18](#).)

Note:

Beginning with the W-2004.12 release, running the `create_ilm` command removes the `is_interface_logic` attribute from the cells, nets, and pins that were identified as part of an interface logic model for the current design.

If you use the `-traverse_disabled_arcs` option with the `create_ilm` command, the disabled timing arcs and paths of a block are ignored, and any interface logic belonging to these disabled arcs or paths is included in the model. Conversely, if you do not use this option, this logic is not included in the model.

Controlling the Fanins and Fanouts of Chip-level Networks

Usually you do not want the fanouts and fanins of chip-level networks, such as scan enable and set/reset networks, to be part of the interface logic model. If you want to automatically exclude the input ports whose nets fanout to a “large” percentage of the total number of registers in the design (defined with the user-defined `ilm_ignore_percentage` variable, default of 25), you can use the `-auto_ignore` option. If you want to exclude specific ports or both input and output ports, you must use the `-ignore_ports` options and manually provide a ports list. Note that these options are mutually exclusive; you can use only one of them.

Because ignored ports have no internal connections in the interface logic model, design rule checking and fixing of the outside connections to these ports tends to be inaccurate. This might require manual intervention to fix. For accurate design rule checking and fixing, use the `-keep_boundary_cells` option to include the boundary cells of all ignored ports in the model.

Controlling the Number of Latch Levels

By default, all transparent latches are identified as part of the interface logic and are assumed to be potential time borrowers. Path tracing continues from the input ports through these latches until an edge-triggered register is encountered. Similarly, path tracing continues from the output registers through the latches to the output ports. You can control the number of latch levels included in the model by using the `-latch_level` option with the `create_ilm` command.

[Figure 10-3](#) shows an interface logic model default model and latch levels.

Figure 10-3 Interface Logic Model Default Model and Latch Levels

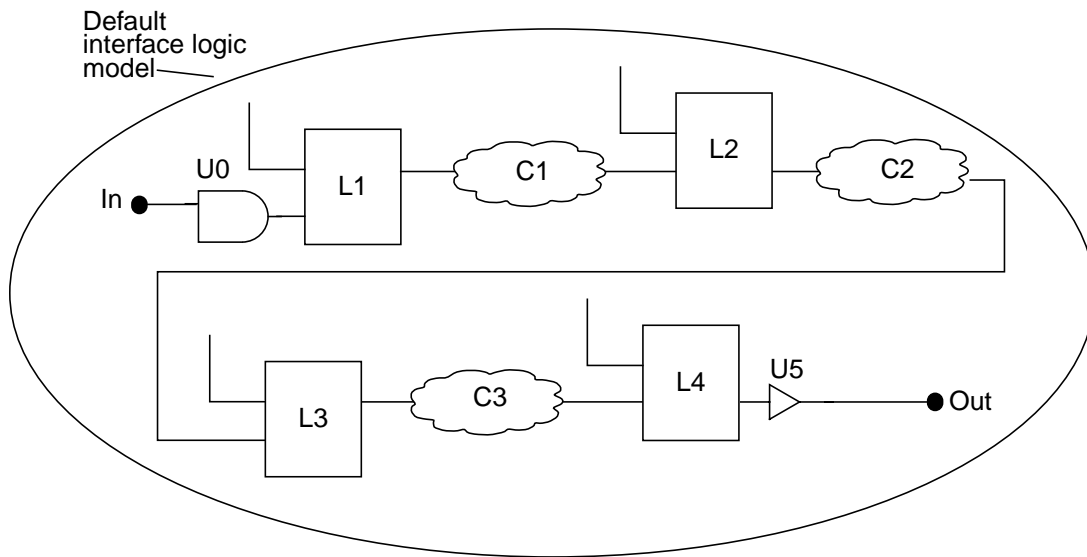
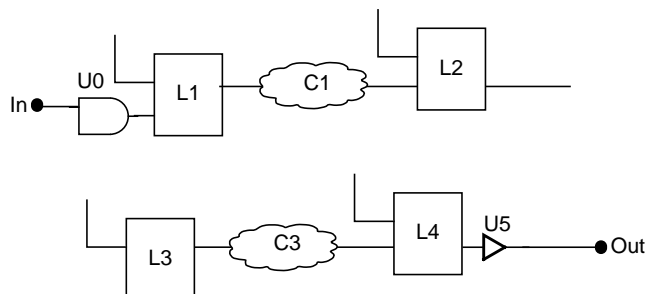


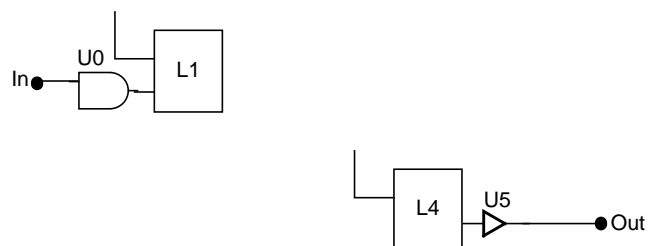
Figure 10-4 shows the results of specifying latch level 1 and latch level 0.

Figure 10-4 Results of Specifying Latch Level 1 and Latch Level 0

Interface logic model
latch levels = 1



Interface logic model
latch levels = 0

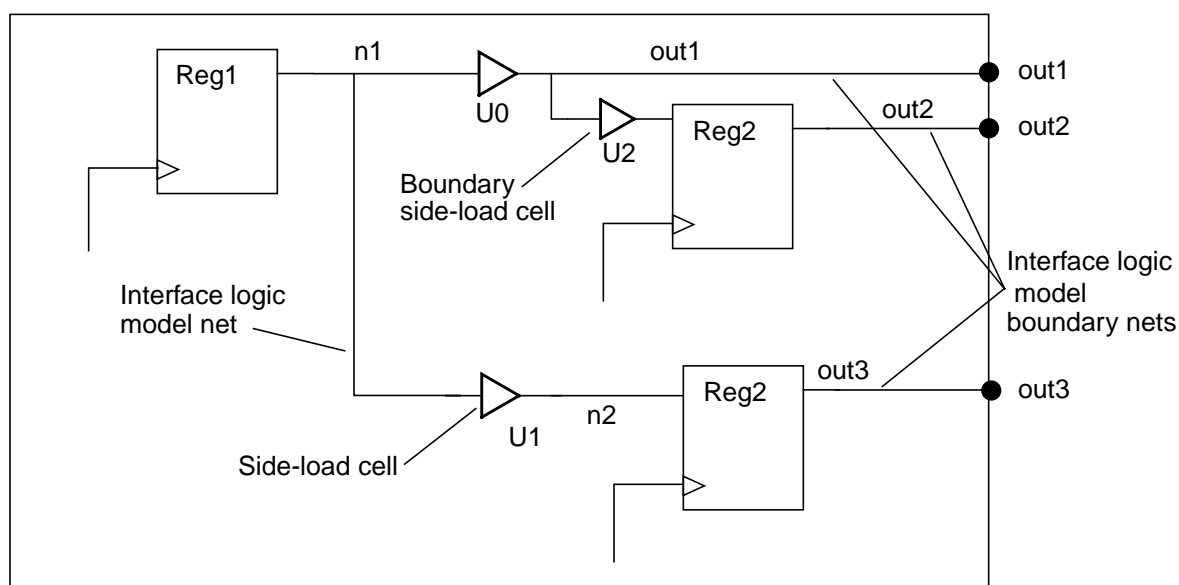


Controlling Side-load Cells

By default, only side-load cells that are boundary cells are included in an interface logic model. These are cells on nets connected to the ports of the model. Even though these cells are not involved in any interface logic timing paths, they might affect the timing of an interface path.

Figure 10-5 shows a side-load cell and a boundary side-load cell.

Figure 10-5 Side-Load Cell and Boundary Side-Load Cell



In [Figure 10-5](#), cell U1 is a side-load cell because it places a load capacitance on the interface logic net n1. Cell U2 is a boundary side-load cell because it places a load capacitance on the boundary interface logic model net out1. Net out1 is considered a boundary net because it is directly connected to a port (out1).

Using the create_ilm Command Options

To use the `create_ilm` command with the appropriate options, enter

dc_shell-t> **create_ilm** [options]

To do this	Use this
Only identify interface logic elements but not build the ILM. Note: The next six options help control what elements are identified as interface logic elements and can be included in the ILM. You can use these options without the <code>-identify_only</code> option or with it.	<code>-identify_only</code>
Exclude the fanout and fanin logic from input and output of ports connected to chip-level nets (for example, scan enable and reset) to get the smallest interface logic model.	<code>-ignore_ports</code>
Exclude automatically the fanout logic from input ports connected to a certain percentage or greater of the total number of registers in the design.	<code>-auto_ignore</code>
Specify the number of latch levels for which time borrowing can occur for latch chains that are part of the interface logic.	<code>-latch_level</code>
Include all macros.	<code>-keep_macros</code>
Include interface logic from disabled timing arcs and pins	<code>-traverse_disabled_arcs</code>
Include the boundary cells of ignored ports.	<code>-keep_boundary_cells</code>
Only extract and create an ILM from currently identified interface logic elements. Note: The next two options help control the extraction process and amount of printed information. You can use these options without the <code>-extract_only</code> option or with it.	<code>-extract_only</code>
Include side-load cells of a specified type.	<code>-include_side_load</code>
Print statistics about the number of design objects in the original design and the model netlist.	<code>-verbose</code>

Note:

You must use the `write` command if you want to save an interface logic model to a file. See [“Saving Interface Logic Models” on page 10-21](#).

For the complete syntax, see the man pages; see also [“Reporting Information About Interface Logic Models” on page 10-27](#).

For scripts you can use to create ILMs, see [“Sample Scripts to Create an Interface Logic Model” on page 10-24](#).

Saving Interface Logic Models

You must issue the `write -f db -hier -output model.db` command to save the model to a file.

Because you must use the `write` command to write the ILM in memory to a file, it is possible to write out an incorrect ILM in the case when the `create_ilm` command did not complete successfully.

Therefore, you should take one of the following precautions before using the `write` command.

- Check the logfile to make sure the process that creates an ILM (`create_ilm`) introduced no errors
- Check the return status of the command you used (`create_ilm`) to create the ILM. (If the command returned a 1, then the command succeeded.)

This is preferred method of checking because it can be scripted.

Note on Changed Names for Common Subdesigns

It is possible to create ILM designs from one or more original designs that have subdesigns in common. However, these common subdesigns, while sharing the same design name, might actually contain different objects (that is, cells, nets, and pins). Therefore, the common subdesign would be common in name only and not in content, because the process that created the ILMs was applied to two (or more) parent designs.

In releases prior to Design Compiler version 2003.12, this situation led to problems, because only one subdesign was linked during the linking process. Typically, you would see warnings similar to the following:

```
Warning: Design 'php_ilm.db:sfifo_W8_DP4_test_1' comes
before design 'risc_ilm.db:sfifo_W8_DP4_test_1' in the
link_library; 'risc_ilm.db:sfifo_W8_DP4_test_1' will be
ignored. (UIO-92)
```

```
Information: Design 'sfifo_W8_DP4_test_1' is referenced in
design 'risc_ilm.db:risc_bus_if_test_1'. (UIO-93)
```

```
Information: Design 'sfifo_W8_DP4_test_1' is referenced in
design 'php_ilm.db:php_interface_test_1'. (UIO-93)
```

1

Then, when you ran the `propagate_ilm` command, the problem resulting from the common subdesigns that are actually different in content leads to log file warnings similar to these:

Warning: Unable to find cell a_core/risc/risc_bus_if/
data_wr_fifo/U80. Attributes will not be loaded.

Warning: Unable to find cell a_core/risc/risc_bus_if/
data_wr_fifo/U79. Attributes will not be loaded.

Warning: Unable to find cell a_core/risc/risc_bus_if/
data_wr_fifo/U78. Attributes will not be loaded.

Warning: Unable to find cell a_core/risc/risc_bus_if/
data_wr_fifo/U77. Attributes will not be loaded.

Warning: Unable to find cell a_core/risc/risc_bus_if/
data_wr_fifo/U76. Attributes will not be loaded

...

The preceding warnings indicate that these cells are in one (common) subdesign but not in the subdesign referenced by the second ILM design.

The solution to this problem that is implemented in Design Compiler version 2003.12 consists of automatically finding and renaming any common subdesigns that exist in memory before linking the designs. For each common subdesign, the renaming process results in a unique subdesign name for the particular ILM that references it. This is accomplished by prepending the ILM .db name to the original (common) subdesign name.

For example, suppose design A is a common subdesign for both files ilm1.db and ilm2.db, that is, ilm1.db:A and ilm2.db:A. The new design names become ilm1_db_A and ilm2_db_A, that is, files now list designs ilm1.db:ilm1_db_A and ilm2.db:ilm2_db_A, which are unique design names.

Here is an example of actual output:

Warning: ILM design or sub-design sfifo_W8_DP4_test_1 in file php_ilm.db has same name as another design. Renamed it as php_ilm_db_sfifo_W8_DP4_test_1. (ILM-50)

Warning: Instance 'inst_fifo' in design 'php_interface_test_1' is updated to refer to renamed design 'php_ilm_db_sfifo_W8_DP4_test_1'. (ILM-51)

Warning: ILM design or sub-design sfifo_W8_DP4_test_1 in file risc_ilm.db has same name as another design. Renamed it as risc_ilm_db_sfifo_W8_DP4_test_1. (ILM-50)

Warning: Instance 'data_wr_fifo' in design 'risc_bus_if_test_1' is updated to refer to renamed design 'risc_ilm_db_sfifo_W8_DP4_test_1'. (ILM-51)

Note:

Although instances will now refer to the modified design names, *the instance names are not changed*. This ensures that all your constraints remain valid.

Sample Scripts to Create an Interface Logic Model

[Example 10-1](#) shows a dcsh script for creating an interface logic model, and [Example 10-2](#) shows the corresponding dctl script. Run one of these scripts (depending on your shell mode) within dc_shell for each model you create.

Example 10-1 Creating an Interface Logic Model (dcsh Mode)

```
/* Create an interface logic model in Design Compiler */
/* Read the compiled block.db file for the block */
read_file block.db
current_design block
link

/* Ensure that the block.db file being read has all clocks defined */
/* Constraints are optional */
/* Optionally apply back-annotation on the block (postroute flow) */
/* Speed the sourcing of the back-annotation data scripts by disabling
   autolinking */
auto_link_disable = true
include block_constraints.scr
include block_set_load.scr
read_sdf block.sdf
auto_link_disable = false

/* Create the interface logic model and write the model to a file */
create_ilm -verbose
write -f db -output block_ilm.db
```

Example 10-2 Creating an Interface Logic Model (dctcl Mode)

```
# Create an interface logic model in Design Compiler
# Read the compiled block.db file for the block
read_db block.db
current_design block
link

# Ensure that the block.db file being read has all clocks defined
# Constraints are optional
# Optionally apply back-annotation on the block (postroute flow)
# Speed the sourcing of the back-annotation data scripts by disabling autolinking

set auto_link_disable true
source block_constraints.tcl
source block_set_load.tcl
read_sdf block.sdf
set auto_link_disable false

# Create the interface logic model and write the model to a file.
create_ilm -verbose
write -f db -output block_ilm.db
```

Example 10-3 shows a script that demonstrates how to include only the lower-level ILM logic that is involved in the timing paths in the upper-level ILM when you have nested ILMs.

Example 10-3 ILM Script: Creating an Upper-Level ILM With Nested ILMs

```
# Read in the ILM block.
read_db ./ilm/block1.db
# Read in the module for which the block1 ILM is a cell
# in the block2 design.
read_verilog ./verilog/block2.v
current_design block2

link

# Propagate up the block1 ILM delay information.
propagate_ilm -delay

create_clock -period 10 [get_ports clk]

compile

# Save the hierarchical design without the ILMs.
foreach_in_collection block [get_ilms -reference] {
    remove_design $block
}

write -f db -hier -out block2_no_ILMs.db

# Read in block1 ILM db.
read_db ./ilm/block1.db

###
# Treat block1 as a logic block so that only the logic in block1
# that is involved in the interface timing paths of block2 will get
# included in the ILM that you create for block2. If you do not
# do this, all the logic of block1 will be included in the
#ILM for block2.
###
current_design block1
set_dont_touch [current_design] false

# Now create an ILM for block2.
current_design block2
create_ilm -verbose
write -f db -hier -output ilm/block2.db
```

Reporting Information About Interface Logic Models

The commands and options listed in [Table 10-1](#) provide information about interface logic models.

Table 10-1 Commands and Options That Report Information About Interface Logic Models

To report this	Use this
Information that the design is an interface logic model.	<code>report_design</code>
Statistics for both the original netlist and the interface logic model netlist to let you know how much reduction occurred for each of the design objects and the total area reduction for interface logic model designs.	<code>report_area</code>
The objects in the current design that are identified as belonging to interface logic.	<code>get_ilm_objects</code>
The interface logic model blocks that are identified as part of the current design.	<code>get_ilms</code>
You can prevent this command from issuing messages by using the <code>-quiet</code> option.	
Statistics for the original design and the interface logic model.	<code>create_ilm -verbose</code>

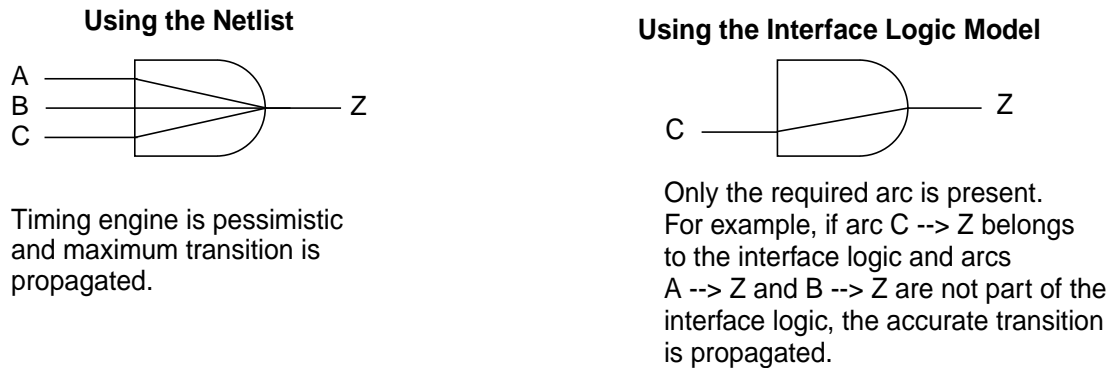
For the complete syntaxes, see the man pages.

Discrepancies between the original design and its interface logic model can arise because of the following:

- Whether or not you include all boundary side loads in the interface logic model. This choice affects the total capacitance on a node and the path timing.

- Propagation of the actual cell interface timing arc instead of the worst-case, maximum transition, as shown in [Figure 10-6](#).

Figure 10-6 Comparison of Timing Propagation Between Netlist and Model



Reporting Design Information

The `report_design` command reports that the design is an interface if you are at the top of the interface logic model. Note the differences in the report results depending on the current design.

- If you have an interface logic model called `ILM_TOP` and `ILM_TOP` is the current design, the report provides the information that it is an interface logic model.
- If you have an interface logic model called `ILM_TOP` that has instances A, B, and C in it and you push into instance A and make it the current design, running `report_design` on design A reports that design A is a subdesign inside the interface logic model design.

Reporting Area Information

The `report_area` command provides statistics for both the original netlist and the interface logic model netlist. These statistics let you know how much area reduction occurred for each of the design objects and for the total area of the interface logic model designs. However, if instance A is the current design, when you run `report_area` on it, you see only the statistics for the interface logic model of instance A.

Compare the reports provided in [Example 10-4](#) and [Example 10-5](#).

Example 10-4 Pre-ILM-Creation Report Area Result When Instance A Is the Current Design

```
*****
Report : area
Design : A
Version: 2002.05
Date   : Mon Apr 1 06:48:14 2002
*****

Library(s) Used:

    fd_inv (File: /remote/stellar04/testcases/129022/lib/fd_inv.db)

Number of ports:      3
Number of nets:      18
Number of cells:      16
Number of references:  2

Combinational area:   100.349998
Noncombinational area: 689.920044
Net Interconnect area: undefined (Wire load has zero net area)

Total cell area:      790.270020
Total area:           undefined
1
```

Example 10-5 Post-ILM-Creation Report Area Result When Instance A Is the Current Design

```
*****
Report : area
Design : A
Version: 2002.05
Date   : Mon Apr 1 06:48:14 5002
*****

Library(s) Used:

    fd_inv (File: /remote/stellar04/testcases/129022/lib/fd_inv.db)

Number of ports:      3
Number of nets:      8
Number of cells:      7
Number of references: 2

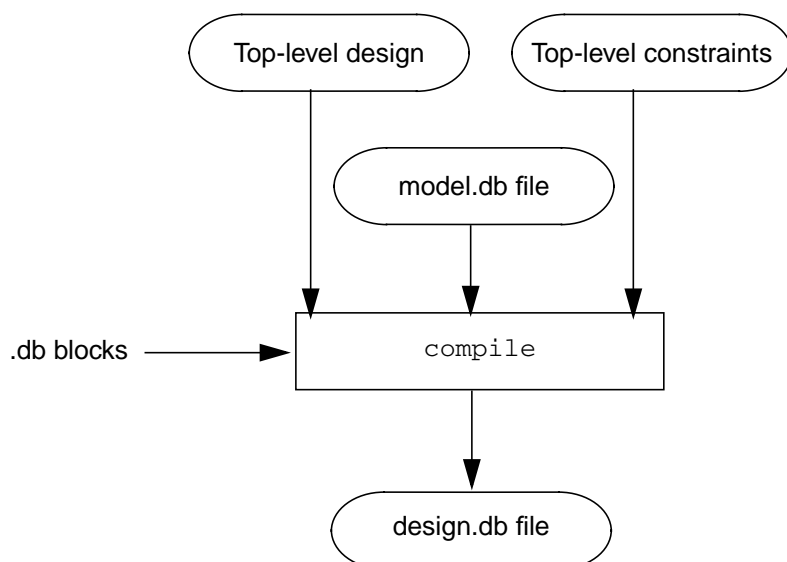
Combinational area:   100.349998
Noncombinational area: 125.440002
Net Interconnect area: undefined (Wire load has zero net area)

Total cell area:      225.790009
Total area:           undefined
1
```

Instantiating and Using Interface Logic Models in Logical Synthesis

After you have generated interface logic models for the block designs you want to replace, you instantiate the models instead of the blocks and compile the top-level design. [Figure 10-7](#) shows the flow for this process.

Figure 10-7 Flow for Instantiating and Using Interface Logic Models



The flow for instantiating interface logic models and using them follows these steps:

1. Read in the top-level design. Use the `read_db` command.
2. Remove the block.db files for the subdesigns to be replaced by the interface logic models (block_ilm.db files) of the subdesigns. Use the `remove_design` command.
3. Read in the interface logic model for each block. Use the `read_db` command.
4. Set the current design to the top-level design.
5. Read in the top-level constraints.
6. Link the design, with the top-level design as the current design.
7. (Optional.) Use the back-annotated data that was applied to the original block by propagating the data up. Use the `propagate_ilm -delay` command.

8. Run logical synthesis, using the interface logic models. Use the `compile` command.
9. Save the design with the interface logic model. Use the `write` command, specifying hierarchy and `.db` format.

Sample Script to Instantiate Interface Logic Models and Run Logical Synthesis in a Hierarchical Route Flow

[Example 10-6](#) shows a `dcsh` script for creating an interface logic model. [Example 10-7](#) shows a `dctl` script for creating an interface logic model. Run one of these scripts (depending on your shell mode) within `dc_shell`.

Example 10-6 Running Top-Level Synthesis (dcsh Mode)

```
/* Instantiate the interface logic models and run logical synthesis */
read_file top.db
remove_design block
read_file block_ilm.db
current_design top
include top_constraints.scr
link

/* If you want to use the back-annotated data applied to the */
/* block, propagate it up to the design top */
propagate_ilm -delay

/* Run logical synthesis using the interface logic models */
compile

/* Save the top-level design */
write -format db -output top_level.db
```


Example 10-7 Running Top-Level Synthesis (dctcl Mode)

```
# Instantiate the interface logic models and run logical synthesis.
read_db top.db
remove_design block
read_db block_ilm.db
current_design top
source top_constraints.tcl
link

# If you want to use the back-annotated data applied to the
# block, propagate it up to the design top.
propagate_ilm -delay

# Run logical synthesis using the interface logic models.
compile

# Save the top-level design.
write -format db -output top_level.db
```

Back-Annotation With Interface Logic Models

Keep in mind the following when using back-annotation with interface logic models:

- Back-annotated data that was applied to a block that became an interface logic model must be propagated up to the top level.
- In order to make the model context independent, back-annotated boundary cell and boundary net delays are not propagated up to the top-level design. This is the default behavior of the `propagate_ilm` command.

Applying Back-Annotation Data for Interface Logic Models

Interface logic models can contain postroute back-annotation data. The back-annotation data consists of SDF information and a script containing `set_load` commands. The SDF can contain net delays and cell delays. The `set_load` annotation is required. Annotating only SDF values can cause the delay and load values to be reestimated.

The back-annotation data can be applied

- To the original block
- To the interface logic model as the current design
- To the interface logic model as a subdesign of the top-level design

In the first two methods, you use the `propagate_ilm` or `propagate_ilm -delay` command to propagate cell delays, pin-to-pin delays, net capacitances, and net resistances to the top-level design.

By default, the annotated boundary cell delays are not propagated to the top level. Instead, interface logic model boundary cell delays are recomputed by the tool when `compile` is run at the top level. You can, however, override this default behavior by using either the `-include_all_boundary_cell_delays` or the `-include_boundary_cell_delays [cell_list]` option. Note that the delays for boundary nets are also not propagated up (default behavior of the `propagate_ilm` command) so that the model will be context independent.

Applying Back-Annotated Data on the Original Block

This method is the preferred method.

Run the `read_sdf` command and `set_load` script on the original block from which an interface logic model will be created. In this case the back-annotation data is applied on the original block-level design. Then, create the interface logic model again for the block-level design. This interface logic model will contain the back-annotated data. When you instantiate the interface logic model in the top-level design, propagate the annotated delays to the top-level current design by using the `propagate_ilm` or `propagate_ilm -delay` command.

Applying Back-Annotation Data on the Interface Logic Model Set as the Current Design

Create the interface logic model and run the `read_sdf` command and `set_load` script with the interface logic model as the current design. In this case the back-annotation data is applied directly on the interface logic model. When you instantiate the interface logic model in the top-level design, propagate the annotated delays to the top-level current design by running the `propagate_ilm` or `propagate_ilm -delay` command.

If the SDF and `set_load` script were generated for the original block, using these annotations on an interface logic model produces error and warning messages for the cells and nets that were removed in the process of creating the model.

Applying Back-Annotation Data on the Interface Logic Model Subdesign With the Current Design as the Top-Level Design

Use the `read_sdf` and `set_load` script on the interface logic model subdesign with the current design set as the top-level design.

If you use this approach, do *not* use the `propagate_ilm -delay` command, because it overwrites the back-annotated data with the data contained in the interface logic model.

If you push into an interface logic model, making it the current design, you will see different timing numbers (output by a `report_timing` command) because the back-annotation values that were applied from the top-level design (when it was the current design) are not pushed into the interface logic model when it is the current design. For the application of back-annotation information from the top level, the interface logic model by itself does not have the correct timing information.

Handling Interface Logic Model Nets That Have Back-Annotated Data

For nets that cross the interface logic model boundary, the delay and load are reestimated based on the complete net connectivity. The tool discards back-annotation data applied to boundary nets and automatically recomputes the values. Internal nets of the model retain the back-annotated delays and load values.

Handling Interface Logic Model Cells That Have Back-Annotated Data

Except for boundary cells, back-annotated cell delays are retained and used without recomputation by the tool. You can include the boundary cell delays by using the

`-include_all_boundary_cell_delays` or the `-include_boundary_cell_delays [cell_list]` option with the `propagate_ilm` command.

Summary of Commands for Interface Logic Models

[Table 10-2](#) lists commands for interface logic models. For further information, see the appropriate sections in this chapter and the man pages.

Table 10-2 Summary of Commands for Interface Logic Models

Command	Description
<code>create_ilm</code>	Determines the objects in the current design that are identified as belonging to interface logic, creates an interface logic model and saves it in memory in .db format. See “Using the create_ilm Command Options” on page 10-19.
<code>propagate_ilm -delay</code>	Propagates annotated delay information from specified reference designs to the current design. See “Back-Annotation With Interface Logic Models” on page 10-33.

Table 10-2 Summary of Commands for Interface Logic Models (Continued)

Command	Description
<code>get_ilms</code>	Returns a collection of interface logic models defined in the current design. Use the <code>-quiet</code> option to prevent this command from issuing messages. See “Reporting Information About Interface Logic Models” on page 10-27 .
<code>get_ilm_objects</code>	Returns a collection of nets, cells, or pins that are identified as belonging to interface logic. See “Reporting Information About Interface Logic Models” on page 10-27 .

A

Design Example

Optimizing a design can involve using different compile strategies for different levels and components in the design. This appendix shows a design example that uses several compile strategies. Earlier chapters provide detailed descriptions of how to implement each compile strategy. Note that the design example used in this appendix does not represent a real-life application.

This appendix includes the following sections:

- [Design Description](#)
- [Setup File](#)
- [Default Constraints File](#)
- [Compile Scripts](#)

You can access the files described in these sections at `$SYNOPTSYS/doc/syn/guidelines`.

Design Description

The design example shows how you can constrain designs by using a subset of the commonly used `dc_shell` commands and how you can use scripts to implement various compile strategies.

The design uses synchronous RTL and combinational logic with clocked D flip-flops.

Figure A-1 shows the block diagram for the design example. The design contains seven modules at the top level: Adder16, CascadeMod, Comparator, Multiply8x8, Multiply16x16, MuxMod, and PathSegment.

Figure A-1 Block Diagram for the Design Example

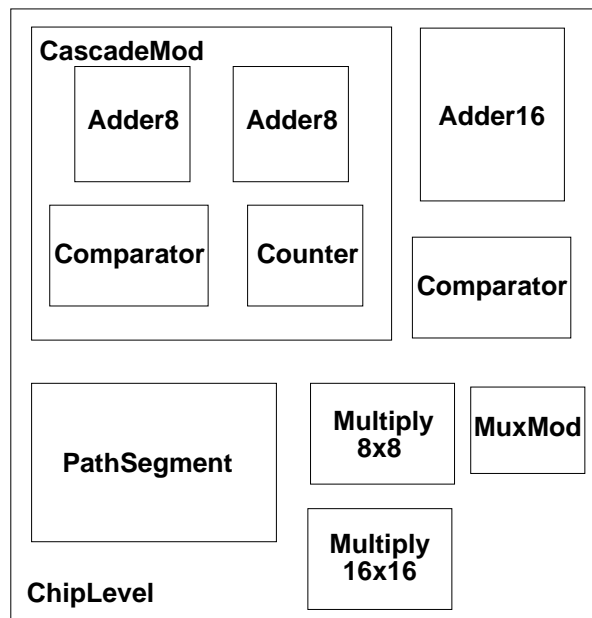
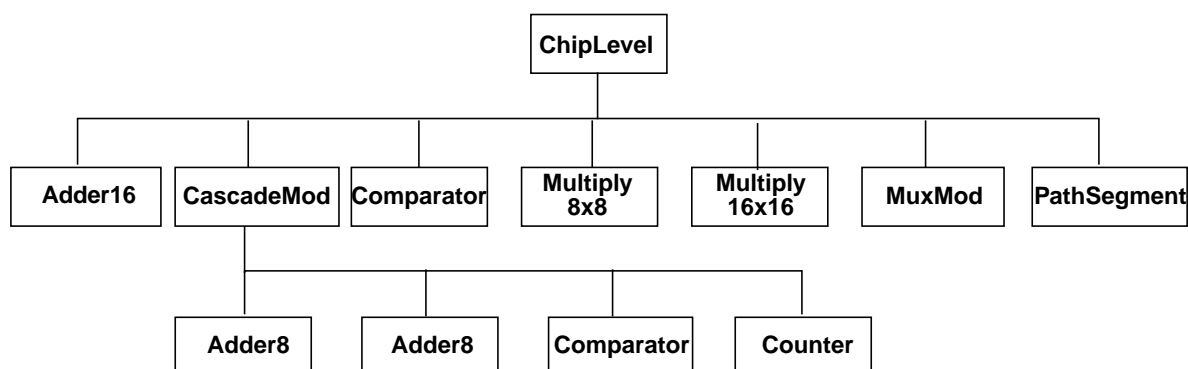


Figure A-2 shows the hierarchy for the design example.

Figure A-2 Hierarchy for the Design Example



The top-level modules and the compilation strategies for optimizing them are

Adder16

Uses registered outputs to make constraining easier. Because the endpoints are the data pins of the registers, you do not need to set output delays on the output ports.

CascadeMod

Uses a hierarchical compile strategy. The compile script for this design sets the constraints at the top level (of CascadeMod) before compilation.

The CascadeMod design instantiates the Adder8 design twice. The script uses the compile-once-don't-touch method for the Comparator module.

Comparator

Is a combinational block. The compile script for this design uses the virtual clock concept to show the use of virtual clocks in a design.

The ChipLevel design instantiates Comparator twice. The compile script (for CascadeMod) uses the compile-once-don't-touch method to resolve the multiple instances.

The compile script specifies wire load model and mode instead of using automatic wire load selection.

Multiply8x8

Shows the basic timing and area constraints used for optimizing a design.

Multiply16x16

Ungroups DesignWare parts before compilation. Ungrouping your hierarchical module might help achieve better synthesis results. The compile script for this module defines a two-cycle path at the primary ports of the module.

MuxMod

Is a combinational block. The script for this design uses the virtual clock concept.

PathSegment

Uses path segmentation within a module. The script uses the `set_multicycle_path` command for a two-cycle path within the module and the `group` command to create a new level of hierarchy.

[Example A-1](#) through [Example A-11](#) provide the Verilog source code for the ChipLevel design.

Example A-1 ChipLevel.v

```
/* Date: May 11, 1995 */
/* Example Circuit for Baseline Methodology for Synthesis */
/* Design does not show any real-life application but rather
   it is used to illustrate the commands used in the Baseline
   Methodology */

module ChipLevel (data16_a, data16_b, data16_c, data16_d, clk, cin, din_a,
                  din_b, sel, rst, start, mux_out, cout1, cout2, s1, s2, op,
                  comp_out1, comp_out2, m32_out, regout);

    input [15:0] data16_a, data16_b, data16_c, data16_d;
    input [7:0] din_a, din_b;
    input [1:0] sel;
    input clk, cin, rst, start;
    input s1, s2, op;
    output [15:0] mux_out, regout;
    output [31:0] m32_out;
    output cout1, cout2, comp_out1, comp_out2;

    wire [15:0] ad16_sout, ad8_sout, m16_out, cnt;

    Adder16 u1 (.ain(data16_a), .bin(data16_b), .cin(cin), .cout(cout1),
               .sout(ad16_sout), .clk(clk));

    CascadeMod u2 (.data1(data16_a), .data2(data16_b), .cin(cin), .s(ad8_sout),
                  .cout(cout2), .clk(clk), .comp_out(comp_out1), .cnt(cnt),
                  .rst(rst), .start(start) );

    Comparator u3 (.ain(ad16_sout), .bin(ad8_sout), .cp_out(comp_out2));

    Multiply8x8 u4 (.op1(din_a), .op2(din_b), .res(m16_out), .clk(clk));

    Multiply16x16 u5 (.op1(data16_a), .op2(data16_b), .res(m32_out), .clk(clk));

    MuxMod u6 (.Y_IN(mux_out), .MUX_CNT(sel), .D(ad16_sout), .R(ad8_sout),
              .F(m16_out), .UPC(cnt));

    PathSegment u7 (.R1(data16_a), .R2(data16_b), .R3(data16_c), .R4(data16_d),
                   .S2(s2), .S1(s1), .OP(op), .REGOUT(regout), .clk(clk));
endmodule
```

Example A-2 Adder16.v

```
module Adder16 (ain, bin, cin, sout, cout, clk);
/* 16-Bit Adder Module */
output [15:0] sout;
output cout;
input [15:0] ain, bin;
input cin, clk;

wire [15:0] sout_tmp, ain_tmp, bin_tmp;
wire cout_tmp;
reg [15:0] sout, ain_tmp, bin_tmp;
reg cout, cin_tmp;

always @(posedge clk) begin
    cout = cout_tmp;
    sout = sout_tmp;
    ain_tmp = ain;
    bin_tmp = bin;
    cin_tmp = cin;
end
    assign {cout_tmp,sout_tmp} = ain_tmp + bin_tmp + cin_tmp;
endmodule
```

Example A-3 CascadeMod.v

```
module CascadeMod (data1, data2, s, clk, cin, cout, comp_out, cnt, rst, start);
input [15:0] data1, data2;
output [15:0] s, cnt;
input clk, cin, rst, start;
output cout, comp_out;
wire co;

Adder8 u10 (.ain(data1[7:0]), .bin(data2[7:0]), .cin(cin), .clk(clk),
    .sout(s[7:0]), .cout(co));
Adder8 u11 (.ain(data1[15:8]), .bin(data2[15:8]), .cin(co), .clk(clk),
    .sout(s[15:8]), .cout(cout));
Comparator u12 (.ain(s), .bin(cnt), .cp_out(comp_out));

Counter u13 (.count(cnt), .start(start), .clk(clk), .rst(rst));
endmodule
```

Example A-4 Adder8.v

```
module Adder8 (ain, bin, cin, sout, cout, clk);
/* 8-Bit Adder Module */
output [7:0] sout;
output cout;
input [7:0] ain, bin;
input cin, clk;

wire [7:0] sout_tmp, ain, bin;
wire cout_tmp;
reg [7:0] sout, ain_tmp, bin_tmp;
reg cout, cin_tmp;

always @(posedge clk) begin
    cout = cout_tmp;
    sout = sout_tmp;
    ain_tmp = ain;
    bin_tmp = bin;
    cin_tmp = cin;
end
    assign {cout_tmp,sout_tmp} = ain_tmp + bin_tmp + cin_tmp;
endmodule
```

Example A-5 Counter.v

```
module Counter (count, start, clk, rst);
/* Counter module */
    input clk;
    input rst;
    input start;
    output [15:0] count;

    wire clk;
    reg [15:0] count_N;
    reg [15:0] count;

    always @ (posedge clk or posedge rst)
        begin : counter_S
            if (rst) begin
                count = 0; // reset logic for the block
            end
            else begin
                count = count_N; // set specified registers of the block
            end
        end

    always @ (count or start)
        begin : counter_C
            count_N = count; // initialize outputs of the block
            if (start) count_N = 1; // user specified logic for the block
            else count_N = count + 1;
        end
    end
endmodule
```

Example A-6 Comparator.v

```
module Comparator (cp_out, ain, bin);
/* Comparator for 2 integer values */
    output cp_out;
    input [15:0] ain, bin;
    assign cp_out = ain < bin;
endmodule
```

Example A-7 Multiply8x8.v

```
module Multiply8x8 (op1, op2, res, clk);
/* 8-Bit multiplier */
input [7:0] op1, op2;
output [15:0] res;
input clk;

wire [15:0] res_tmp;
reg [15:0] res;

always @(posedge clk) begin
    res = res_tmp;
end
assign res_tmp = op1 * op2;
endmodule
```

Example A-8 Multiply16x16.v

```
module Multiply16x16 (op1, op2, res, clk);
/* 16-Bit multiplier */
input [15:0] op1, op2;
output [31:0] res;
input clk;

wire [31:0] res_tmp;
reg [31:0] res;

always @(posedge clk) begin
    res = res_tmp;
end
assign res_tmp = op1 * op2;
endmodule
```

Example A-9 def_macro.v

```
`define DATA 2'b00
`define REG 2'b01
`define STACKIN 2'b10
`define UPCOUT 2'b11
```

Example A-10 MuxMod.v

```
module MuxMod (Y_IN, MUX_CNT, D, R, F, UPC);
`include "def_macro.v"
    output [15:0] Y_IN;
    input [ 1:0] MUX_CNT;
    input [15:0] D, F, R, UPC;

    reg [15:0] Y_IN;

    always @ ( MUX_CNT or D or R or F or UPC ) begin
        case ( MUX_CNT )
            `DATA :
                Y_IN = D ;
            `REG :
                Y_IN = R ;
            `STACKIN :
                Y_IN = F ;
            `UPCOUT :
                Y_IN = UPC;
        endcase
    end

endmodule
```


Example A-11 PathSegment.v

```
module PathSegment (R1, R2, R3, R4, S2, S1, OP, REGOUT, clk);
/* Example for path segmentation */
input [15:0] R1, R2, R3, R4;
input S2, S1, clk;
input OP;
output [15:0] REGOUT;

reg [15:0] ADATA, BDATA;
reg [15:0] REGOUT;
reg MODE;

wire [15:0] product ;

always @(posedge clk)
begin : selector_block
    case(S1)
        1'b0: ADATA <= R1;
        1'b1: ADATA <= R2;
        default: ADATA <= 16'bx;
    endcase
    case(S2)
        1'b0: BDATA <= R3;
        1'b1: BDATA <= R4;
        default: ADATA <= 16'bx;
    endcase
end

/* Only Lower Byte gets multiplied */
// instantiate DW02_mult
DW02_mult #(8,8) U100 (.A(ADATA[7:0]), .B(BDATA[7:0]), .TC(1'b0),
.PRODUCT(product));

always @(posedge clk)
begin : alu_block
    case (OP)
        1'b0 : begin
            REGOUT <= ADATA + BDATA;
        end
        1'b1 : begin
            REGOUT <= product;
        end
        default : REGOUT <= 16'bx;
    endcase
end

endmodule
```

Setup File

When running the design example, copy the project-specific setup file in [Example A-12](#) to your project working directory. This setup file is written in the Tcl subset and can be used in both dcsh and dctcl mode. For more information about the Tcl subset, see the *Design Compiler Command-Line Interface Guide*.

Important:

If you are using dcsh mode and the setup file in your home directory uses dcsh syntax instead of the Tcl subset, this setup file will not work.

For details on the synthesis setup files, see [“Setup Files” on page 2-8](#).

Example A-12 `.synopsys_dc.setup` File

```
# Define the target technology library, symbol library,
# and link libraries
set target_library lsi_10k.db
set symbol_library lsi_10k.sdb
set link_library [concat $target_library "*"]
set search_path [concat $search_path ./src]
set designer "Your Name"
set company "Synopsys, Inc."
# Define path directories for file locations
set source_path "./src/"
set script_path "./scr/"
set log_path "./log/"
set db_path "./db/"
set netlist_path "./netlist/"
```

Default Constraints File

The file shown in [Example A-13](#) (dcsh mode) and [Example A-14](#) (dctcl mode) defines the default constraints for the design. In the scripts that follow, Design Compiler reads this file first for each module. If the script for a module contains additional constraints or constraint values different from those defined in the default constraints file, Design Compiler uses the module-specific constraints.

Example A-13 defaults.con (dcsh Mode)

```
/* Define system clock period*/
clk_period = 20

/* Create real clock if clock port is found */
if (find(port, clk) == {"clk"}) {
    clk_name = clk
    create_clock -period clk_period clk
}

/* Create virtual clock if clock port is not found */
if (find(port, clk) == {}) {
    clk_name = vclk
    create_clock -period clk_period -name vclk
}

/* Apply default drive strengths and typical loads
for I/O ports */
set_load 1.5 all_outputs()
set_driving_cell -cell IV all_inputs()

/* If real clock, set infinite drive strength */
if (find(port, clk) == {"clk"}) {
    set_drive 0 clk
}

/* Apply default timing constraints for modules */
set_input_delay 1.2 all_inputs() -clock clk_name
set_output_delay 1.5 all_outputs() -clock clk_name
set_clock_uncertainty -setup 0.45 clk_name

/* Set operating conditions */
set_operating_conditions WCCOM

/* Turn on auto wire load selection (library must support
this feature) */
auto_wire_load_selection = true
```

Example A-14 defaults.con (dctcl Mode)

```
# Define system clock period
set clk_period 20

# Create real clock if clock port is found
if {[sizeof_collection [get_ports clk]] > 0} {
    set clk_name clk
    create_clock -period $clk_period clk
}

# Create virtual clock if clock port is not found
if {[sizeof_collection [get_ports clk]] == 0} {
    set clk_name vclk
    create_clock -period $clk_period -name vclk
}

# Apply default drive strengths and typical loads
# for I/O ports
set_load 1.5 [all_outputs]
set_driving_cell -cell IV [all_inputs]

# If real clock, set infinite drive strength
if {[sizeof_collection [get_ports clk]] > 0} {
    set_drive 0 clk
}

# Apply default timing constraints for modules
set_input_delay 1.2 [all_inputs] -clock $clk_name
set_output_delay 1.5 [all_outputs] -clock $clk_name
set_clock_uncertainty -setup 0.45 $clk_name

# Set operating conditions
set_operating_conditions WCCOM

# Turn on auto wire load selection
# (library must support this feature)
set auto_wire_load_selection true
```

Compile Scripts

[Example A-15](#) through [Example A-26](#) provide the dcsh scripts used to compile the ChipLevel design. [Example A-27](#) through [Example A-38](#) provide the dctcl scripts used to compile the ChipLevel design.

The compile script for each module is named for that module to ease recognition. The initial dcsh script files have the .scr suffix. The initial dctcl script files have the .tcl suffix. Scripts generated by the `write_script` command have the .wscr or .wtcl suffix, depending on which mode generated them.

Example A-15 run.scr

```
/* Initial compile with estimated constraints */
include script_path + initial_compile.scr

current_design ChipLevel
write -hierarchy -out db_path + ChipLevel_init.db

/* Characterize and write_script for all modules */
include script_path + characterize.scr

/* Recompile all modules using write_script constraints */
remove_design -all
include script_path + recompile.scr

current_design ChipLevel
write -hierarchy -out db_path + ChipLevel_final.db
```

Example A-16 initial_compile.scr

```
/* Initial compile with estimated constraints */
include script_path + read.scr

current_design ChipLevel
include script_path + defaults.con

include script_path + adder16.scr
include script_path + cascademod.scr
include script_path + comp16.scr
include script_path + mult8.scr
include script_path + mult16.scr
include script_path + muxmod.scr
include script_path + pathseg.scr
```

Example A-17 adder16.scr

```
/* Script file for constraining Adder16 */
rpt_file = "adder16.rpt"
design = "adder16"

current_design Adder16
include script_path + defaults.con

/* Define design environment */
set_load 2.2 sout
set_load 1.5 cout
set_driving_cell -cell FD1 all_inputs()
set_drive 0 clk_name

/* Define design constraints */
set_input_delay 1.35 -clock clk_name {ain, bin}
set_input_delay 3.5 -clock clk_name cin
set_max_area 0

compile
write -f db -hierarchy -o db_path + design + ".db"
include script_path + report.scr
```

Example A-18 cascademod.scr

```
/* Script file for constraining CascadeMod */
/* Constraints are set at this level and then a hierarchical
compile approach is used */

rpt_file = "cascademod.rpt"
design = "cascademod"

current_design CascadeMod
include script_path + defaults.con

/* Define design environment */
set_load 2.5 all_outputs()
set_driving_cell -cell FD1 all_inputs()
set_drive 0 clk_name

/* Define design constraints */
set_input_delay 1.35 -clock clk_name {data1, data2}
set_input_delay 3.5 -clock clk_name cin
set_input_delay 4.5 -clock clk_name {rst, start}
set_output_delay 5.5 -clock clk_name comp_out
set_max_area 0

/* Use compile-once, dont_touch approach for Comparator */
set_dont_touch u12

compile
write -f db -hierarchy -o db_path + design + ".db"
include script_path + report.scr
```


Example A-19 comp16.scr

```
/* Script file for constraining Comparator */
rpt_file = "comp16.rpt"
design = "comp16"

current_design Comparator
include script_path + defaults.con

/* Define design environment */
set_load 2.5 cp_out
set_driving_cell -cell FD1 all_inputs()

/* Override auto wire load selection */
set_wire_load_model -name "05x05"
set_wire_load_mode enclosed

/* Define design constraints */
set_input_delay 1.35 -clock clk_name {ain, bin}
set_output_delay 5.1 -clock clk_name {cp_out}
set_max_area 0

compile
write -f db -hierarchy -o db_path + design + ".db"
include script_path + report.scr
```

Example A-20 mult8.scr

```
/* Script file for constraining Multiply8x8 */
rpt_file = "mult8.rpt"
design = "mult8"

current_design Multiply8x8
include script_path + defaults.con

/* Define design environment */
set_load 2.2 res
set_driving_cell -cell FD1P all_inputs()
set_drive 0 clk_name

/* Define design constraints */
set_input_delay 1.35 -clock clk_name {op1, op2}
set_max_area 0

compile
write -f db -hierarchy -o db_path + design + ".db"
include script_path + report.scr
```

Example A-21 *mult16.scr*

```
/* Script file for constraining Multiply16x16 */
rpt_file = "mult16.rpt"
design = "mult16"

current_design Multiply16x16
include script_path + defaults.con

/* Define design environment */
set_load 2.2 res
set_driving_cell -cell FD1 all_inputs()
set_drive 0 clk_name

/* Define design constraints */
set_input_delay 1.35 -clock clk_name {op1, op2}
set_max_area 0

/* Define multicycle path for multiplier */
set_multicycle_path 2 -from all_inputs() \
    -to all_registers(-data_pins -edge_triggered)

/* Ungroup DesignWare parts */
designware_cells = \
    filter( find(cell), "@is_oper==true" )
if (dc_shell_status != {}) {
    set_ungroup designware_cells true
}

compile
write -f db -hierarchy -o db_path + design + ".db"
include script_path + report.scr
report_timing_requirements -ignore \
    >> log_path + rpt_file
```

Example A-22 muxmod.scr

```
/* Script file for constraining MuxMod */
rpt_file = "muxmod.rpt"
design = "muxmod"

current_design MuxMod
include script_path + defaults.con

/* Define design environment */
set_load 2.2 Y_IN
set_driving_cell -cell FD1 all_inputs()

/* Define design constraints */
set_input_delay 1.35 -clock clk_name {D, R, F, UPC}
set_input_delay 2.35 -clock clk_name MUX_CNT
set_output_delay 5.1 -clock clk_name {Y_IN}
set_max_area 0

compile
write -f db -hierarchy -o db_path + design + ".db"
include script_path + report.scr
```

Example A-23 *pathseg.scr*

```
/* Script file for constraining path_segment */
rpt_file = "pathseg.rpt"
design = "pathseg"

current_design PathSegment
include script_path + defaults.con

/* Define design environment */
set_load 2.5 all_outputs()
set_driving_cell -cell FD1 all_inputs()
set_drive 0 clk_name

/* Define design rules */
set_max_fanout 6 {S1 S2}

/* Define design constraints */
set_input_delay 2.2 -clock clk_name {R1 R2}
set_input_delay 2.2 -clock clk_name {R3 R4}
set_input_delay 5 -clock clk_name {S2 S1 OP}
set_max_area 0

/* Perform path segmentation for multiplier */
group -design mult -cell mult U100
set_input_delay 10 -clock clk_name mult/product*
set_output_delay 5 -clock clk_name mult/product*
set_multicycle_path 2 -to mult/product*

compile
write -f db -hierarchy -o db_path + design + ".db"
include script_path + report.scr
report_timing_requirements -ignore \
    >> log_path + rpt_file
```

Example A-24 *characterize.scr*

```
/* Characterize and write_script for all modules */
current_design ChipLevel
characterize u1
current_design Adder16
write_script > script_path + adder16.wscr

current_design ChipLevel
characterize u2
current_design CascadeMod
write_script > script_path + cascademod.wscr

current_design ChipLevel
characterize u3
current_design Comparator
write_script > script_path + comp16.wscr

current_design ChipLevel
characterize u4
current_design Multiply8x8
write_script > script_path + mult8.wscr

current_design ChipLevel
characterize u5
current_design Multiply16x16
write_script > script_path + mult16.wscr

current_design ChipLevel
characterize u6
current_design MuxMod
write_script > script_path + muxmod.wscr

current_design ChipLevel
characterize u7
current_design PathSegment
echo "current_design PathSegment" > \
    script_path + pathseg.wscr
echo "group -design mult -cell mult U100" >> \
    script_path + pathseg.wscr
write_script >> script_path + pathseg.wscr
```

Example A-25 recompile.scr

```
include script_path + read.scr

current_design ChipLevel
include script_path + defaults.con

include script_path + adder16.wscr
compile
write -f db -hier -o db_path + adder16_wscr.db
rpt_file = adder16_wscr.rpt
include script_path + report.scr

include script_path + cascademod.wscr
dont_touch u12
compile
write -f db -hier -o db_path + cascademod_wscr.db
rpt_file = cascade_wscr.rpt
include script_path + report.scr

include script_path + compl6.wscr
compile
write -f db -hier -o db_path + compl6_wscr.db
rpt_file = compl6_wscr.rpt
include script_path + report.scr

include script_path + mult8.wscr
compile
write -f db -hier -o db_path + mult8_wscr.db
rpt_file = mult8_wscr.rpt
include script_path + report.scr

include script_path + mult16.wscr
compile -ungroup_all
write -f db -hier -o db_path + mult16_wscr.db
rpt_file = mult16_wscr.rpt
include script_path + report.scr
report_timing_requirements -ignore \
    >> log_path + rpt_file
include script_path + muxmod.wscr
compile
write -f db -hier -o db_path + muxmod_wscr.db
rpt_file = muxmod_wscr.rpt
include script_path + report.scr
```

Example A-25 recompile.scr (Continued)

```
include script_path + pathseg.wscr
compile
write -f db -hier -o db_path + pathseg_wscr.db
rpt_file = pathseg_wscr.rpt
include script_path + report.scr
report_timing_requirements -ignore \
    >> log_path + rpt_file
```

Example A-26 report.scr

```
/* This script file creates reports for all modules */
maxpaths = 15

check_design > log_path + rpt_file
report_area >> log_path + rpt_file
report_design >> log_path + rpt_file
report_cell >> log_path + rpt_file
report_reference >> log_path + rpt_file
report_port -verbose >> log_path + rpt_file
report_net >> log_path + rpt_file
report_compile_options >> log_path + rpt_file
report_constraint -all_violators -verbose \
    >> log_path + rpt_file
report_timing -path end >> log_path + rpt_file
report_timing -max_path maxpaths \
    >> log_path + rpt_file
```


Example A-27 run.tcl

```
# Initial compile with estimated constraints
source "${script_path}initial_compile.tcl"

current_design ChipLevel
write -hierarchy -out "${db_path}ChipLevel_init.db"

# Characterize and write_script for all modules
source "${script_path}characterize.tcl"

# Recompile all modules using write_script constraints
remove_design -all
source "${script_path}recompile.tcl"

current_design ChipLevel
write -hierarchy -out "${db_path}ChipLevel_final.db"
```

Example A-28 initial_compile.tcl

```
# Initial compile with estimated constraints
source "${script_path}read.tcl"

current_design ChipLevel
source "${script_path}defaults.con"

source "${script_path}adder16.tcl"
source "${script_path}cascademod.tcl"
source "${script_path}comp16.tcl"
source "${script_path}mult8.tcl"
source "${script_path}mult16.tcl"
source "${script_path}muxmod.tcl"
source "${script_path}pathseg.tcl"
```

Example A-29 adder16.tcl

```
# Script file for constraining Adder16
set rpt_file "adder16.rpt"
set design "adder16"

current_design Adder16
source "${script_path}defaults.con"

# Define design environment
set_load 2.2 sout
set_load 1.5 cout
set_driving_cell -cell FD1 [all_inputs]
set_drive 0 $clk_name

# Define design constraints
set_input_delay 1.35 -clock $clk_name {ain bin}
set_input_delay 3.5 -clock $clk_name cin
set_max_area 0

compile
write -f db -hierarchy -o "${db_path}${design}.db"
source "${script_path}report.tcl"
```

Example A-30 cascademod.tcl

```
# Script file for constraining CascadeMod
# Constraints are set at this level and then a
# hierarchical compile approach is used

set rpt_file "cascademod.rpt"
set design "cascademod"

current_design CascadeMod
source "${script_path}defaults.con"

# Define design environment
set_load 2.5 [all_outputs]
set_driving_cell -cell FD1 [all_inputs]
set_drive 0 $clk_name

# Define design constraints
set_input_delay 1.35 -clock $clk_name {data1 data2}
set_input_delay 3.5 -clock $clk_name cin
set_input_delay 4.5 -clock $clk_name {rst start}
set_output_delay 5.5 -clock $clk_name comp_out
set_max_area 0

# Use compile-once, dont_touch approach for Comparator
set_dont_touch u12

compile
write -f db -hierarchy -o "${db_path}${design}.db"
source "${script_path}report.tcl"
```

Example A-31 comp16.tcl

```
# Script file for constraining Comparator
set rpt_file "comp16.rpt"
set design "comp16"

current_design Comparator
source "${script_path}defaults.con"

# Define design environment
set_load 2.5 cp_out
set_driving_cell -cell FD1 [all_inputs]

# Override auto wire load selection
set_wire_load_model -name "05x05"
set_wire_load_mode enclosed

# Define design constraints
set_input_delay 1.35 -clock $clk_name {ain bin}
set_output_delay 5.1 -clock $clk_name {cp_out}
set_max_area 0

compile
write -f db -hierarchy -o "${db_path}${design}.db"
source "${script_path}report.tcl"
```

Example A-32 mult8.tcl

```
# Script file for constraining Multiply8x8
set rpt_file "mult8.rpt"
set design "mult8"

current_design Multiply8x8
source "${script_path}defaults.con"

# Define design environment
set_load 2.2 res
set_driving_cell -cell FD1P [all_inputs]
set_drive 0 $clk_name

# Define design constraints
set_input_delay 1.35 -clock $clk_name {op1 op2}
set_max_area 0

compile
write -f db -hierarchy -o "${db_path}${design}.db"
source "${script_path}report.tcl"
```

Example A-33 *mult16.tcl*

```
# Script file for constraining Multiply16x16
set rpt_file "mult16.rpt"
set design "mult16"

current_design Multiply16x16
source "${script_path}defaults.con"

# Define design environment
set_load 2.2 res
set_driving_cell -cell FD1 [all_inputs]
set_drive 0 $clk_name

# Define design constraints
set_input_delay 1.35 -clock $clk_name {op1 op2}
set_max_area 0

# Define multicycle path for multiplier
set_multicycle_path 2 -from [all_inputs] \
    -to [all_registers -data_pins -edge_triggered]

# Ungroup DesignWare parts
set designware_cells [get_cells \
    -filter "@is_oper==true"]
if {[sizeof_collection $designware_cells] > 0} {
    set_ungroup $designware_cells true
}

compile
write -f db -hierarchy -o "${db_path}${design}.db"
source "${script_path}report.tcl"
report_timing_requirements -ignore \
    >> "${log_path}${rpt_file}"
```

Example A-34 muxmod.tcl

```
# Script file for constraining MuxMod
set rpt_file "muxmod.rpt"
set design "muxmod"

current_design MuxMod
source "${script_path}defaults.con"

# Define design environment
set_load 2.2 Y_IN
set_driving_cell -cell FD1 [all_inputs]

# Define design constraints
set_input_delay 1.35 -clock $clk_name {D R F UPC}
set_input_delay 2.35 -clock $clk_name MUX_CNT
set_output_delay 5.1 -clock $clk_name {Y_IN}
set_max_area 0

compile
write -f db -hierarchy -o "${db_path}${design}.db"
source "${script_path}report.tcl"
```

Example A-35 pathseg.tcl

```
# Script file for constraining path_segment
set rpt_file "pathseg.rpt"
set design "pathseg"

current_design PathSegment
source "${script_path}defaults.con"

# Define design environment
set_load 2.5 [all_outputs]
set_driving_cell -cell FD1 [all_inputs]
set_drive 0 $clk_name

# Define design rules
set_max_fanout 6 {S1 S2}

# Define design constraints
set_input_delay 2.2 -clock $clk_name {R1 R2}
set_input_delay 2.2 -clock $clk_name {R3 R4}
set_input_delay 5 -clock $clk_name {S2 S1 OP}
set_max_area 0

# Perform path segmentation for multiplier
group -design mult -cell mult U100
set_input_delay 10 -clock $clk_name mult/product*
set_output_delay 5 -clock $clk_name mult/product*
set_multicycle_path 2 -to mult/product*

compile
write -f db -hierarchy -o "${db_path}${design}.db"
source "${script_path}report.tcl"
report_timing_requirements -ignore \
    >> "${log_path}${rpt_file}"
```


Example A-36 characterize.tcl

```
# Characterize and write_script for all modules
current_design ChipLevel
characterize u1
current_design Adder16
write_script > "${script_path}adder16.wtcl"

current_design ChipLevel
characterize u2
current_design CascadeMod
write_script -format dctcl > \
    "${script_path}cascademod.wtcl"

current_design ChipLevel
characterize u3
current_design Comparator
write_script -format dctcl > "${script_path}comp16.wtcl"

current_design ChipLevel
characterize u4
current_design Multiply8x8
write_script -format dctcl > "${script_path}mult8.wtcl"

current_design ChipLevel
characterize u5
current_design Multiply16x16
write_script -format dctcl > "${script_path}mult16.wtcl"

current_design ChipLevel
characterize u6
current_design MuxMod
write_script -format dctcl > "${script_path}muxmod.wtcl"

current_design ChipLevel
characterize u7
current_design PathSegment

echo "current_design PathSegment" > \
    "${script_path}pathseg.wtcl"

echo "group -design mult -cell mult U100" >> \
    "${script_path}pathseg.wtcl"
write_script -format dctcl >> "${script_path}pathseg.wtcl"
```

Example A-37 *recompile.tcl*

```
source "${script_path}read.tcl"

current_design ChipLevel
source "${script_path}defaults.con"

source "${script_path}adder16.wtcl"
compile
write -f db -hier -o "${db_path}adder16_wtcl.db"
set rpt_file adder16_wtcl.rpt
source "${script_path}report.tcl"

source "${script_path}cascademod.wtcl"
dont_touch u12
compile
write -f db -hier -o "${db_path}cascademod_wtcl.db"
set rpt_file cascade_wtcl.rpt
source "${script_path}report.tcl"

source "${script_path}comp16.wtcl"
compile
write -f db -hier -o "${db_path}comp16_wtcl.db"
set rpt_file comp16_wtcl.rpt
source "${script_path}report.tcl"

source "${script_path}mult8.wtcl"
compile
write -f db -hier -o "${db_path}mult8_wtcl.db"
set rpt_file mult8_wtcl.rpt
source "${script_path}report.tcl"

source "${script_path}mult16.wtcl"
compile -ungroup_all
write -f db -hier -o "${db_path}mult16_wtcl.db"
set rpt_file mult16_wtcl.rpt
source "${script_path}report.tcl"
report_timing_requirements -ignore \
    >> "${log_path}${rpt_file}"
source "${script_path}muxmod.wtcl"
compile
write -f db -hier -o "${db_path}muxmod_wtcl.db"
set rpt_file muxmod_wtcl.rpt
source "${script_path}report.tcl"
```

Example A-37 recompile.tcl (Continued)

```
source "${script_path}pathseg.wtcl"
compile
write -f db -hier -o "${db_path}pathseg_wtcl.db"
set rpt_file pathseg_wtcl.rpt
source "${script_path}report.tcl"
report_timing_requirements -ignore \
    >> "${log_path}${rpt_file}"
```

Example A-38 report.tcl

```
# This script file creates reports for all modules
set maxpaths 15
```

```
check_design > "${log_path}${rpt_file}"
report_area >> "${log_path}${rpt_file}"
report_design >> "${log_path}${rpt_file}"
report_cell >> "${log_path}${rpt_file}"
report_reference >> "${log_path}${rpt_file}"
report_port -verbose >> "${log_path}${rpt_file}"
report_net >> "${log_path}${rpt_file}"
report_compile_options >> "${log_path}${rpt_file}"
report_constraint -all_violators -verbose \
    >> "${log_path}${rpt_file}"
report_timing -path end >> "${log_path}${rpt_file}"
report_timing -max_path $maxpaths \
    >> "${log_path}${rpt_file}"
```


B

Basic Commands

This appendix lists the basic `dc_shell` commands for synthesis and provides a brief description for each command. The commands are grouped in the following sections:

- [Commands for Defining Design Rules](#)
- [Commands for Defining Design Environments](#)
- [Commands for Setting Design Constraints](#)
- [Commands for Analyzing and Resolving Design Problems](#)

Within each section the commands are listed in alphabetical order.

Commands for Defining Design Rules

The commands that define design rules are

`set_max_capacitance`

Sets a maximum capacitance for the nets attached to the specified ports or to all the nets in a design.

`set_max_fanout`

Sets the expected fanout load value for output ports.

`set_max_transition`

Sets a maximum transition time for the nets attached to the specified ports or to all the nets in a design.

`set_min_capacitance`

Sets a minimum capacitance for the nets attached to the specified ports or to all the nets in a design.

Commands for Defining Design Environments

The commands that define the design environment are

`set_drive`

Sets the drive value of input or inout ports. The `set_drive` command is superseded by the `set_driving_cell` command.

`set_driving_cell`

Sets attributes on input or inout ports, specifying that a library cell or library pin drives the ports. This command associates a library pin with an input port so that delay calculators can accurately model the drive capability of an external driver.

`set_fanout_load`

Defines the external fanout load values on output ports.

`set_load`

Defines the external load values on input and output ports and nets.

`set_operating_conditions`

Defines the operating conditions for the current design.

`set_wire_load_model`

Sets the wire load model for the current design or for the specified ports. With this command, you can specify the wire load model to use for the external net connected to the output port.

Commands for Setting Design Constraints

The basic commands that set design constraints are

`create_clock`

Creates a clock object and defines its waveform in the current design.

`set_clock_latency, set_clock_uncertainty,`
`set_propagated_clock, set_clock_transition`

Sets clock attributes on clock objects or flip-flop clock pins.

`set_input_delay`

Sets input delay on pins or input ports relative to a clock signal.

`set_max_area`

Specifies the maximum area for the current design.

`set_output_delay`

Sets output delay on pins or output ports relative to a clock signal.

The advanced commands that set design constraints are

`group_path`

Groups a set of paths or endpoints for cost function calculation. This command is used to create path groups, to add paths to existing groups, or to change the weight of existing groups.

`set_false_path`

Marks paths between specified points as false. This command eliminates the selected paths from timing analysis.

`set_max_delay`

Specifies a maximum delay target for selected paths in the current design.

`set_min_delay`

Specifies a minimum delay target for selected paths in the current design.

`set_multicycle_path`

Allows you to specify the time of a timing path to exceed the time of one clock signal.

Commands for Analyzing and Resolving Design Problems

The commands for analyzing and resolving design problems are

`all_connected`

Lists all fanouts on a net.

`all_registers`

Lists sequential elements or pins in a design.

`check_design`

Checks for violations of the connection class rules; displays a list of warning messages when violations exist.

`check_timing`

Checks the timing attributes placed on the current design.

`derive_clocks`

Automatically creates clocks for all clock sources in a design; creates clock objects on ports and internal sources.

`get_attribute`

Reports the value of the specified attribute.

`link`

Locates the reference for each cell in the design.

`report_area`

Provides area information and statistics on the current design.

`report_attribute`

Lists the attributes and their values for the selected object. An object can be a cell, net, pin, port, instance, or design.

`report_cell`

Lists the cells in the current design and their cell attributes.

`report_clock`

Displays clock-related information on the current design.

`report_constraint`

Lists the constraints on the current design and their cost, weight, and weighted cost.

`report_delay_calculation`

Reports the details of a delay arc calculation.

`report_design`

Displays the operating conditions, wire load model and mode, timing ranges, internal input and output, and disabled timing arcs defined for the current design.

`report_hierarchy`

Lists the children of the current design.

`report_net`

Displays net information for the design of the current instance, if set; otherwise, displays net information for the current design.

`report_path_group`

Lists all timing path groups in the current design.

`report_port`

Lists information about ports in the current design.

`report_qor`

Displays information about the quality of results and other statistics for the current design.

`report_resources`

Displays information about the resource implementation.

`report_timing`

Lists timing information for the current design.

`report_timing_requirements`

Lists timing path requirements and related information.

`report_transitive_fanin`

Lists the fanin logic for selected pins, nets, or ports of the current instance.

`report_transitive_fanout`

Lists the fanout logic for selected pins, nets, or ports of the current instance.

C

Predefined Attributes

This appendix contains tables that list the Design Compiler predefined attributes for each object type.

Table C-1 Clock Attributes

Attribute name	Value
dont_touch_network	{true, false}
fall_delay	float
fix_hold	{true, false}
max_time_borrow	float
minus_uncertainty	float
period	float
plus_uncertainty	float
propagated_clock	{true, false}

Table C-1 Clock Attributes (Continued)

Attribute name	Value
rise_delay	float

Table C-2 Design Attributes

Attribute name	Value
actual_max_net_capacitance	float
actual_min_net_capacitance	float
boundary_optimization	{true, false}
default_flip_flop_type	internally generated string
default_flip_flop_type_exact	library_cell_name
default_latch_type	library_cell_name
design_type	{equation, fsm, pla, netlist}
dont_touch	{true, false}
dont_touch_network	{true, false}
driven_by_logic_one	{true, false}
driven_by_logic_zero	{true, false}
driving_cell_dont_scale	string
driving_cell_fall	string
driving_cell_from_pin_fall	string
driving_cell_from_pin_rise	string
driving_cell_library_fall	string
driving_cell_library_rise	string
driving_cell_multiplier	float

Table C-2 Design Attributes (Continued)

Attribute name	Value
driving_cell_pin_fall	string
driving_cell_pin_rise	string
driving_cell_rise	string
fall_drive	float
fanout_load	float
flatten	{true, false}
flatten_effort	{true, false}
flatten_minimize	{true, false}
flatten_phase	{true, false}
flip_flop_type	internally generated string
flip_flop_type_exact	library_cell_name
is_black_box	{true, false}
is_combinational	{true, false}
is_hierarchical	{true, false}
is_mapped	{true, false}
is_sequential	{true, false}
is_test_circuitry	{true, false}
is_unmapped	{true, false}
latch_type	internally generated string
latch_type_exact	library_cell_name
load	float

Table C-2 Design Attributes (Continued)

Attribute name	Value
local_link_library	design_or_lib_file_name
max_capacitance	float
max_fanout	float
max_time_borrow	float
max_transition	float
min_capacitance	float
minus_uncertainty	float
output_not_used	{true, false}
pad_location (XNF only)	string
part (XNF only)	string
plus_uncertainty	float
port_direction	{in, inout, out, unknown}
port_is_pad	{true, false}
ref_name	reference_name
rise_drive	float
structure	{true, false}
ungroup	{true, false}
wired_logic_disable	{true, false}
xnf_init	string
xnf_loc	string

Table C-3 Library Attributes

Attribute name	Value
default_values	float
k_process_values	float
k_temp_values	float
k_volt_values	float
nom_process	float
nom_temperature	float
nom_voltage	float

Table C-4 Library Cell Attributes

Attribute name	Value
area	float
dont_touch	{true, false}
dont_use	{true, false}
preferred	{true, false}

Table C-5 Net Attributes

Attribute name	Value
ba_net_resistance	float
dont_touch	{true, false}
load	float
subtract_pin_load	{true, false}
wired_and	{true, false}

Table C-5 Net Attributes (Continued)

Attribute name	Value
wired_or	{true, false}

Table C-6 Pin Attributes

Attribute name	Value
disable_timing	{true, false}
max_time_borrow	float
pin_direction	{in, inout, out, unknown}

Table C-7 Reference Attributes

Attribute name	Value
dont_touch	{true, false}
is_black_box	{true, false}
is_combinational	{true, false}
is_hierarchical	{true, false}
is_mapped	{true, false}
is_sequential	{true, false}
is_unmapped	{true, false}
ungroup	{true, false}

Glossary

annotation

A piece of information attached to an object in the design, such as a capacitance value attached to a net; the process of attaching such a piece of information to an object in the design.

back-annotate

To update a circuit design by using extraction and other post-processing information that reflects implementation-dependent characteristics of the design, such as pin selection, component location, or parasitic electrical characteristics. Back-annotation allows a more accurate timing analysis of the final circuit. The data is generated by another tool after layout and passed to the synthesis environment. For example, the design database might be updated with actual interconnect delays; these delays are calculated after placement and routing—after exact interconnect lengths are known.

cell

See instance.

clock

A source of timed pulses with a periodic behavior. A clock synchronizes the propagation of data signals by controlling sequential elements, such as flip-flops and registers, in a digital circuit. You define clocks with the `create_clock` command.

Clocks you create by using the `create_clock` command ignore delay effects of the clock network. Therefore, for accurate timing analysis, you describe the clock network in terms of its latency and skew. See also clock latency and clock skew.

clock gating

The control of a clock signal by logic (other than inverters or buffers), either to shut down the clock signal at selected times or to modify the clock pulse characteristics.

clock latency

The amount of time that a clock signal takes to be propagated from the clock source to a specific point in the design. Clock latency is the sum of source latency and network latency.

Source latency is the propagation time from the actual clock origin to the clock definition point in the design. Network latency is the propagation time from the clock definition point in the design to the clock pin of the first register.

You use the `set_clock_latency` command to specify clock latency.

clock skew

The maximum difference between the arrival of clock signals at registers in one clock domain or between clock domains. Clock skew is also known as clock uncertainty. You use the `set_clock_uncertainty` command to specify the skew characteristics of one or more clock networks.

clock source

The pin or port where the clock waveform is applied to the design. The clock signal reaches the registers in the transitive fanout of all its sources. A clock can have multiple sources.

You use the `create_clock` command with the `source_object` option to specify clock sources.

clock tree

The combinational logic between a clock source and registers in the transitive fanout of that source. Clock trees, also known as clock networks, are synthesized by vendors based on the physical placement data at registers in one clock domain or between clock domains.

clock uncertainty

See clock skew.

core

A predesigned block of logic employed as a building block for ASIC designs.

critical path

The path through a circuit with the longest delay. The speed of a circuit depends on the slowest register-to-register delay. The clock period cannot be shorter than this delay or the signal will not reach the next register in time to be clocked.

datapath

A logic circuit in which data signals are manipulated using arithmetic operators such as adders, multipliers, shifters, and comparators.

current design

The active design (the design being worked on). Most commands are specific to the current design, that is, they operate within the context of the current design. You specify the current design with the `current_design` command.

current instance

The instance in a design hierarchy on which instance-specific commands operate by default. You specify the current instance with the `current_instance` command.

design constraints

The designer's specification of design performance goals, that is, the timing and environmental restrictions under which synthesis is to be performed. Design Compiler uses these constraints—for example, low power, small area, high-speed, or minimal cost—to direct the optimization of a design to meet area and timing goals.

There are two categories of design constraints: design rule constraints and design optimization constraints.

- Design rule constraints are supplied in the technology library. For proper functioning of the fabricated circuit, they must not be violated.
- Design optimization constraints define timing and area optimization goals.

Design Compiler optimizes the synthesis of the design in accordance with both sets of constraints; however, design rule constraints have higher priority.

false path

A path that you do not want Design Compiler to consider during timing analysis. An example of such a path is one between two multiplexed blocks that are never enabled at the same time, that is, a path that cannot propagate a signal.

You use the `set_false_path` command to disable timing-based synthesis on a path-by-path basis. The command removes timing constraints on the specified path.

fanin

The pins driving an endpoint pin, port, or net (also called sink). A pin is considered to be in the fanin of a sink if there is a timing path through combinational logic from the pin to the sink. Fanin tracing starts at the clock pins of registers or valid startpoints. Fanin is also known as transitive fanin.

You use the `report_transitive_fanin` command to report the fanin of a specified sink pin, port, or net.

fanout

The pins driven by a source pin, port, or net. A pin is considered to be in the fanout of a source if there is a timing path through combinational logic from the source to that pin or port. Fanout tracing stops at the data pin of a register or at valid endpoints. Fanout is also known as transitive fanout or timing fanout.

You use the `report_transitive_fanout` command to report the fanout of a specified source pin, port, or net.

fanout load

A unitless value that represents a numerical contribution to the total fanout. Fanout load is not the same as load, which is a capacitance value.

Design Compiler models fanout restrictions by associating a `fanout_load` attribute with each input pin and a `max_fanout` attribute with each output (driving) pin on a cell and ensures that the sum of fanout loads is less than the `max_fanout` value.

flatten

To convert combinational logic paths of the design to a two-level, sum-of-products representation. During flattening, Design Compiler removes all intermediate terms, and therefore all associated logic structure, from a design. Flattening is constraint based.

You use the `set_flatten` command to control the flattening of your design. Design Compiler does not perform flattening by default.

forward-annotate

To transfer data from the synthesis environment to other tools used later in the design flow. For example, delay and constraints data in Standard Delay Format (SDF) might be transferred from the synthesis environment to guide place and route tools.

generated clock

A clock signal that is generated internally by the integrated circuit itself; a clock that does not come directly from an external source. An example of a generated clock is a divide-by-2 clock generated from the system clock. You define a generated clock with the `create_generated_clock` command.

hold time

The time that a signal on the data pin must remain stable after the active edge of the clock. The hold time creates a minimum delay requirement for paths leading to the data pin of the cell.

You calculate the hold time by using the formula

$$\text{hold} = \text{max clock delay} - \text{min data delay}$$
ideal clock

A clock that is considered to have no delay as it propagates through the clock network. The ideal clock type is the default for Design Compiler. You can override the default behavior (using the `set_clock_latency` and `set_propagated_clock` commands) to obtain nonzero clock network delay and specify information about the clock network delays.

ideal net

Nets that are assigned ideal timing conditions—that is, latency, transition time, and capacitance are assigned a value of zero. Such nets are exempt from timing updates, delay optimization, and design rule fixing. Defining certain high fanout nets that you intend to synthesize separately (such as scan-enable and reset nets) as ideal nets can reduce runtime.

You use the `set_ideal_net` command to specify nets as ideal nets.

input delay

A constraint that specifies the minimum or maximum amount of delay from a clock edge to the arrival of a signal at a specified input port.

You use the `set_input_delay` command to set the input delay on a pin or input port relative to a specified clock signal.

instance

An occurrence in a circuit of a reference (a library component or design) loaded in memory; each instance has a unique name. A design can contain multiple instances; each instance points to the same reference but has a unique name to distinguish it from other instances. An instance is also known as a cell.

leaf cell

A fundamental unit of logic design. A leaf cell cannot be broken into smaller logic units. Examples are NAND gates and inverters.

link library

A technology library that Design Compiler uses to resolve cell references. Link libraries can contain technology libraries and design files. Link libraries also contain the descriptions of cells (library cells as well as subdesigns) in a mapped netlist.

Link libraries include both local link libraries (`local_link_library` attribute) and system link libraries (`link_library` variable).

multicycle path

A path for which data takes more than one clock cycle to propagate from the startpoint to the endpoint.

You use the `set_multicycle_path` command to specify the number of clock cycles Design Compiler should use to determine when data is required at a particular endpoint.

netlist

A file in ASCII or binary format that describes a circuit schematic—the netlist contains a list of circuit elements and interconnections in a design. Netlist transfer is the most common way of moving design information from one design system or tool to another.

operating conditions

The process, voltage, and temperature ranges a design encounters. Design Compiler optimizes your design according to an operating point on the process, voltage, and temperature curves and scales cell and wire delays according to your operating conditions.

By default, operating conditions are specified in a technology library in an `operating_conditions` group.

optimization

The step in the logic synthesis process in which Design Compiler attempts to implement a combination of technology library cells that best meets the functional, timing, and area requirements of the design.

output delay

A constraint that specifies the minimum or maximum amount of delay from an output port to the sequential element that captures data from the output port. This constraint establishes the times at which signals must be available at the output port to meet the setup and hold requirements of the sequential element.

You use the `set_output_delay` command to set the output delay on a pin or output port relative to a specified clock signal.

pad cell

A special cell at the chip boundaries that allows connection or communication with integrated circuits outside the chip.

path group

A group of related paths, grouped either implicitly by the `create_clock` command or explicitly by the `group_path` command. By default, paths whose endpoints are clocked by the same clock are assigned to the same path group.

pin

A part of a cell that provides for input and output connections. Pins can be bidirectional. The ports of a subdesign are pins within the parent design.

propagated clock

A clock that incurs delay through the clock network. Propagated clocks are used to determine clock latency at register clock pins. Registers clocked by a propagated clock have edge times skewed by the path delay from the clock source to the register clock pin.

You use the `set_propagated_clock` command to specify that clock latency be propagated through the clock network.

real clock

A clock that has a source, meaning its waveform is applied to pins or ports in the design. You create a real clock by using a `create_clock` command and including a source list of ports or pins. Real clocks can be either ideal or propagated.

reference

A library component or design that can be used as an element in building a larger circuit. The structure of the reference may be a simple logic gate or a more complex design (RAM core or CPU). A design can contain multiple occurrences of a reference; each occurrence is an instance. See also `instance`.

RTL

RTL, or register transfer level, is a register-level description of a digital electronic circuit. In a digital circuit, registers store intermediate information between clock cycles; thus, RTL describes the intermediate information that is stored, where it is stored within

the design, and how it is transferred through the design. RTL models circuit behavior at the level of data flow between a set of registers. This level of abstraction typically contains little timing information, except for references to a set of clock edges and features.

setup time

The time that a signal on the data pin must remain stable before the active edge of the clock. The setup time creates a maximum delay requirement for paths leading to the data pin of a cell.

You calculate the setup time by using the formula

$$\text{setup} = \text{max data delay} - \text{min clock delay}$$

slack

A value that represents the difference between the actual arrival time and the required arrival time of data at the path endpoint in a mapped design. Slack values can be positive, negative, or zero.

A positive slack value represents the amount by which the delay of a path can be increased without violating any timing constraints. A negative slack value represents the amount by which the delay of a path must be reduced to meet its timing constraints.

structuring

To add intermediate variables and logic structure to a design, which can result in reduced design area. Structuring is constraint based. It is best applied to noncritical timing paths.

By default, Design Compiler structures your design. You use the `set_structure` command and the `compile_new_boolean_structure` variable to control the structuring of your design.

synthesis

A software process that generates an optimized gate-level netlist, which is based on a technology library, from an input IC design. Synthesis includes reading the HDL source code and optimizing the design from that description.

symbol library

A library that contains the schematic symbols for all cells in a particular ASIC library. Design Compiler uses symbol libraries to generate the design schematic. You can use Design Vision to view the design schematic.

target library

The technology library to which Design Compiler maps during optimization. Target libraries contain the cells used to generate the netlist and definitions for the design's operating conditions.

technology library

A library of ASIC cells that are available to Design Compiler during the synthesis process. A technology library can contain area, timing, power, and functional information on each ASIC cell. The technology of each library is specific to a particular ASIC vendor.

timing exception

An exception to the default (single-cycle) timing behavior assumed by Design Compiler. For Design Compiler to analyze a circuit correctly, you must specify each timing path in the design that does not conform to the default behavior. Examples of timing exceptions include false paths, multicycle paths, and paths that require a specific minimum or maximum delay time different from the default calculated time.

timing path

A point-to-point sequence that dictates data propagation through a design. Data is launched by a clock edge at a startpoint, propagated through combinational logic elements, and captured at an endpoint.

by another clock edge. The startpoint of a timing path is an input port or clock pin of a sequential element. The endpoint of a timing path is an output port or a data pin of a sequential element.

transition delay

A timing delay caused by the time it takes the driving pin to change voltage state.

ungroup

To remove hierarchy levels in a design. Ungrouping merges subdesigns of a given level of the hierarchy into the parent cell or design.

You use the `ungroup` command or the `compile` command with the `auto_ungroup` option to ungroup designs.

uniquify

To resolve multiple cell references to the same design in memory.

You use the `uniquify` command to create unique design copies with unique design names for each instantiated cell that references the original design.

virtual clock

A clock that exists in the system but is not part of the block. A virtual clock does not clock any sequential devices within the current design and is not associated with a pin or port. You use a virtual clock as a reference for specifying input and output delays relative to a clock outside the block.

You use the `create_clock` command without a list of associated pins or ports to create a virtual clock.

wire load model

An estimate of a net's RC parasitics based on the net's fanout, in the absence of placement and routing information. The estimated capacitance and resistance are used to calculate the delay of nets. After placement and routing, you should back-annotate the design with detailed information on the net delay.

The wire load model is shipped with the technology library; vendors develop the wire load model based on statistical information specific to the vendor's process. You can also custom-generate the model based on back-annotation. The model includes coefficients for area, capacitance, and resistance per unit length, and a fanout-to-length table for estimating net lengths (the number of fanouts determines a nominal length).

Appendix GL:

GL-14

Index

A

- accessing help 2-12
- all_clocks command 5-18
- all_connected command 5-39
- all_fanin command 8-49
- all_outputs command 5-18
- all_registers command 5-18
- analyze command 2-20, 5-8
- analyzing design 9-8
- architectural optimization 8-2
- area constraints
 - command to set 7-25
- async_set_reset compiler directive 3-16
- attribute values
 - saving 5-53
 - setting 5-51
 - viewing 5-52
- attributes
 - creating 5-53
 - defined 5-50
 - design rule 7-3
 - dont_touch 10-8
 - getting descriptions 5-51
 - listing 5-31
 - removing 5-31, 5-54, 7-5
 - search order 5-52, 5-53
 - viewing 9-25
- attributes, list of
 - auto_wire_load_selection 6-12
 - cell_degradation 7-3
 - clock C-1
 - connection_class 7-3
 - default_wire_load 6-11
 - default_wire_load_mode 6-12
 - design C-2
 - dont_touch 8-28, 9-16, 9-25, 9-27
 - fanout_load 7-5
 - is_black_box 9-14
 - is_hierarchical 9-14
 - is_unmapped 9-13
 - library C-5
 - library cell C-5
 - max_area 7-25
 - max_capacitance 7-8, 9-26
 - max_fanout 7-5, 9-26
 - max_transition 7-4, 9-26
 - net C-5
 - pin C-6
 - reference C-6
 - signal_type 9-15
- auto_ungroup_preserve_constraints variable 5-34
- auto_wire_load_selection attribute 6-12

B

- balance_buffer command 8-48
- Berkeley Espresso (PLA) format 5-8
- Boolean optimization
 - defined 8-54
 - enabling 8-54
- bottom-up compile 8-12
 - advantages 8-12
 - directory structure
 - figure 3-4
 - disadvantages 8-12
 - process 8-13
 - when to use 8-12
- boundary optimization 8-56
- breaking, feedback loop 9-17
- buffers
 - extra 9-26
 - guidelines for working with 9-23
 - hanging 9-27
 - insertion process 9-17
 - interblock 9-16
 - missing 9-23
- buses
 - creating 5-38
 - deleting 5-38

C

- capacitance
 - calculating 7-8
 - checking 9-24
 - controlling 7-7
 - cost calculation 8-32
 - removing attribute 7-8
- capacitive load
 - setting 6-17
- case sensitive
 - setting 5-15
- case statement 3-24
 - latch inference 3-25
 - multiplexer inference 3-15

- cell count-based auto-ungrouping 5-32
- cell degradation cost 8-33
- cell delays, finding source of 9-11
- cell_degradation attribute 7-3
- cells
 - attributes 5-5
 - black box, identifying 9-14
 - creating 5-38
 - deleting 5-38
 - grouping
 - from different subdesigns 5-37
 - from same subdesign 5-25
 - hierarchical
 - defined 5-5
 - identifying 9-14
 - leaf 5-5
 - library, specifying 4-12
 - listing 5-17
 - merging
 - hierarchy 5-37
 - reporting 5-17
 - unmapped, identifying 9-13
- change_link command 5-16
- change_names command 5-30, 5-45
- check_design command 9-2
- checkpointing
 - automatically 9-7
 - defined 9-5
 - manually 9-6
- cinterface_logic_model
 - creating 10-19
- clock attributes C-1
- clock network delay
 - default 7-12
 - reporting 7-13
 - setting margin of error 7-13
 - specifying 7-12
- clock uncertainty
 - setting 7-10
- clocks
 - creating 7-10

- defining 7-11
- ideal 7-12
- listing 5-18
- multiple 7-11
- removing 7-12
- reporting 5-18, 7-12
- See also, clock network delay
- specifying
 - network delay 7-12
 - period 7-11
 - waveform 7-11
- combinational logic
 - partitioning 3-5
 - specifying delay requirements 7-16
- command
 - analyzing design problems B-5
 - design constraints
 - setting B-3
 - design environment B-2
 - design rules B-2
 - resolving design problems B-5
- command log files 2-13
- command script 2-14
- command_log_file variable 2-14
- commands
 - all_clocks 5-18
 - all_connected 5-39
 - all_fanin 8-49
 - all_outputs 5-18
 - all_registers 5-18
 - analyze 2-20, 5-8
 - analyzing design 9-8
 - balance_buffer 8-48
 - change_link 5-16
 - change_names 5-30, 5-45
 - check_design 9-2
 - compile -auto_ungroup area 5-31
 - compile -auto_ungroup delay 5-31
 - compile_auto_ungroup delay 8-51
 - compile_ultra 8-42
 - connect_net 5-38
 - copy_design 5-22
 - create_bus 5-38
 - create_cell 5-38
 - create_clock 7-10, 7-11, 8-34
 - create_design 5-21
 - create_ilm 10-11
 - create_multibit 3-18
 - create_net 5-35, 5-38
 - create_port 5-36, 5-38
 - current_design 5-13
 - current_instance 5-19
 - define_name_rules -map 5-46
 - disconnect_net 5-38
 - elaborate 2-20, 5-8
 - exit 2-12
 - filter 9-13, 9-14
 - find 9-13, 9-14
 - get_attribute 5-52, 9-24, 9-25
 - get_cells 9-13, 9-14
 - get_designs 9-14
 - get_ilm_objects 10-27
 - get_ilms 10-27
 - get_license 2-16
 - group 5-25, 8-49
 - group_path 8-34, 8-45
 - license_users 2-15
 - list 4-11, 6-12
 - list_designs 5-11
 - list_duplicate_designs 5-12
 - list_instances 5-17
 - list_libs 4-11, 6-9
 - load_of 9-24
 - propagate_ilm 10-33, 10-34, 10-35
 - quit 2-12
 - read_file 2-20, 4-11, 5-8, 5-13
 - read_lib 4-11
 - read_sdf 10-35, 10-36
 - read_verilog 2-20
 - read_vhdl 2-20
 - remove_attribute 7-5, 7-7, 7-8
 - remove_bus 5-38
 - remove_cell 5-38
 - remove_clock 7-12

- remove_constraint 7-26
- remove_design 4-16, 5-42
- remove_input_delay 7-14
- remove_license 2-16
- remove_multibit 3-18
- remove_net 5-38
- remove_output_delay 7-14
- remove_port 5-38
- remove_wire_load_model 6-13
- rename_design 5-23
- report_area 10-27, 10-29
- report_attribute 5-52
- report_auto_ungroup 5-32
- report_cell 9-25
- report_clock 5-18, 7-12
- report_compile_options 8-5
- report_constraint 9-18
- report_delay_calculation 9-11
- report_design 6-5, 8-29, 10-27, 10-28
- report_hierarchy 5-24
- report_lib 6-4, 6-9, 6-17
- report_net 5-18
- report_path_group 8-34
- report_port 5-18, 7-14
- report_reference 5-18
- report_resources 8-70
- report_timing 6-13, 9-17
- report_timing_requirements 7-16, 7-17
- reset_path 7-18, 7-19, 7-23
- set_clock_uncertainty 7-10
- set_cost_priority 8-31
- set_critical_range 8-46
- set_disable_timing 9-14
- set_dont_touch 8-24, 8-28
- set_drive 6-14, 6-15, 6-16
- set_driving_cell 6-13, 6-14, 6-16
- set_false_path 7-17
- set_fanout_load 6-18, 7-6
- set_flatten 8-4
- set_input_delay 7-10, 7-13
- set_input_transition 6-14
- set_load 6-17, 10-5, 10-34, 10-35, 10-36
- set_max_area 7-25
- set_max_delay 7-16, 7-19
- set_max_fanout 7-6
- set_max_transition 7-4
- set_min_delay 7-16, 7-19
- set_multicycle_path 7-21
- set_output_delay 7-10, 7-13
- set_resistance 9-16
- set_structure 8-3, 8-54
- set_ultra_optimization 8-73
- set_ungroup 5-30, 8-57
- set_wire_load 6-7, 6-12
- translate 5-40, 5-41
- ungroup 5-28, 8-26
- uniquify 8-22
- write 5-43
- write_lib 4-16
- write_script 5-53
- common base period, defined 7-11
- compile
 - default 8-39
 - defined 2-2
 - directory structure
 - bottom-up 3-4
 - top-down 3-3
 - high effort 8-51
 - incremental 8-52
- compile -auto_ungroup area 5-31
- compile -auto_ungroup delay 5-31
- compile command
 - automatically uniquified designs 8-21
 - default behavior 8-39
 - disabling design rule cost function 8-31
 - disabling optimization cost function 8-31
- compile cost function 8-30
- compile log
 - customizing 9-3
- compile option
 - list of 8-5
- compile script A-16
 - adder16 A-17, A-28

- cascademod A-18, A-29
- comparator A-19, A-30
- multiply 8x8 A-20, A-31
- multiply16x16 A-21, A-32
- muxmod A-22, A-33
- pathseg A-23, A-34
- compile scripts
 - design example A-16
- compile strategies 8-7
- compile strategy
 - bottom-up 8-12
 - defined 2-22
 - mixed 8-19
 - top-down 8-9
- compile_assume_fully_decoded_three_state_busses variable 5-42
- compile_auto_ungroup_area_num_cells variable 5-32, 5-34
- compile_auto_ungroup_delay_num_cells variable 5-33, 5-34, 8-43
- compile_auto_ungroup_override_wlm 5-34
- compile_auto_ungroup_override_wlm variable 5-34
- compile_fix_cell_degradation variable 8-33
- compile_new_boolean_structure variable 8-54
- compile_sequential_area_recovery variable 8-54
- compile_ultra command 8-42
- compiler directives
 - async_set_reset 3-16
 - enum 3-21
 - full_case 3-25
 - implementation 3-19
 - infer_multibit 3-18
 - infer_mux 3-15
 - label 3-19
 - map_to_module 3-19, 3-32
 - ops 3-19
 - return_port_name 3-32
 - state_vector 3-21
 - sync_set_reset 3-17
- compiler_log_format variable 9-4
- connect_net command 5-38
- connection_class attribute 7-3
- constants
 - global
 - defining 3-26
- constraints
 - area 7-25
 - defining 7-1
 - design rule
 - setting 7-3
 - removing 7-26
 - simplifying 3-7
 - timing 7-9
- constraints file
 - design example A-13
- copy_design command 5-22
- cost calculation
 - capacitance 8-32
 - fanout 8-32
 - maximum delay 8-33
 - minimum delay 8-36
 - minimum porosity 8-38
 - transition time 8-31
- cost function 8-30
 - constraints
 - report_constraint command 9-19
 - design rule 8-30
 - optimization 8-30
- create_bus command 5-38
- create_cell command 5-38
- create_clock command 7-10, 7-12
 - and path groups 8-34
 - clock, defining 7-11
 - default behavior 7-11
- create_design command 5-21
- create_ilm command 10-11
- create_multibit command 3-18
- create_net command 5-35, 5-38
- create_port command 5-36, 5-38
- critical negative slack, defined 8-35

- critical range, defined 8-35
- critical-path resynthesis 8-51

- current design
 - defined 5-5
 - displaying 5-13
 - setting 5-13

- current instance 5-5
 - changing 5-19
 - default 5-19
 - defined 5-18
 - displaying 5-19
 - resetting 5-19

- current_design
 - command 5-13
 - variable 5-13

- current_instance
 - command 5-19
 - variable 5-19

D

- dangling logic, preserving 9-16

- data management 3-2

- data organization 3-3

- datapath extraction

 - DC Ultra 8-60

- datapath optimization

 - DC Ultra 8-58

 - three methods 8-64

- .db (Synopsys internal database) format 5-7

- DB mode 2-7

- DC Expert
 - defined 1-5

- DC FPGA
 - defined 1-8

- DC Ultra
 - defined 1-6

- DC Ultra datapath optimization 8-58

 - bit truncation 8-62

 - commands and variables, specific to 8-73

 - datapath extraction 8-60

- datapath report 8-70

- licenses required 8-59, 8-67

- methodology flow 8-67

- three optimization methods 8-64

- dc_shell

 - exiting 2-11

 - session example 2-24

 - starting 2-10

 - dcsh mode 2-10

 - dctl mode 2-10

- DC-Expert license 8-51

- dcsh mode 2-7

- dctl mode 2-7

- default compile 8-39

- default_wire_load attribute 6-11

- default_wire_load_mode attribute 6-12

- define_name_rules -map command 5-46

- definitions

 - attribute 5-50

 - Boolean optimization 8-54

 - checkpointing 9-5

 - common base period 7-11

 - compiler 2-2

 - critical range 8-35

 - current design 5-5

 - current instance 5-5, 5-18

 - design 5-3

 - flat design 5-3

 - hierarchical cell 5-5

 - hierarchical design 5-3

 - instance 5-5

 - leaf cell 5-5

 - negative slack

 - critical 8-35

 - total 8-35

 - worst 8-33

 - nets 5-7

 - networks 5-7

 - optimization 2-2

 - parent design 5-3

 - pin 5-7

- ports 5-7
- reference 5-6
- subdesign 5-3
- synthesis 2-2
- delay calculation, reporting 9-11
- delay cost, calculating
 - maximum 8-33
 - minimum 8-36
- delay-based auto-ungrouping 5-33
- delays
 - setting 7-10, 7-13
- design
 - data management 3-2
 - in memory 5-1
 - organization 3-3
- design attributes C-2
- Design Compiler
 - description 1-1
 - design flow 1-2
 - exiting 2-11
 - family of products 1-4
 - help 2-12
 - interfaces 2-7
 - modes 2-7
 - session example 2-24
 - starting 2-10
- Design Compiler family
 - DC Expert 1-5
 - DC FPGA 1-8
 - DC Ultra 1-6
 - Design Vision 1-8
 - DesignWare 1-7
 - DFT Compiler 1-7
 - HDL Compiler 1-6
 - Module Compiler 1-7
 - Power Compiler 1-7
- design constraints
 - commands
 - setting B-3
- design environment
 - commands B-2

- defining 6-3
 - See also, operating conditions
- design example
 - block diagram A-2
 - compile scripts A-16
 - compile strategies for A-3
 - constraints file A-13
 - hierarchy A-3
 - setup file A-12
- design exploration 8-39
 - basic flow 2-16
 - invoking 8-39
- design files
 - reading 2-20, 5-8
 - search path for 5-9
 - supported formats 5-7
 - writing 5-43
- design flow 1-2
 - high-level
 - figure 2-5
 - synthesis
 - design exploration 2-16
 - design implementation 2-16
- design function
 - target libraries 4-4
- design hierarchy
 - changing 5-24
 - displaying 5-24
 - preserved timing constraints 5-37
 - removing levels 5-27
 - See also, hierarchy
- design implementation 8-40
 - basic flow 2-16
 - techniques for 8-40
- design objects
 - accessing 5-17
 - adding 5-22
 - defined 5-4
 - listing
 - clocks 5-18
 - instances 5-17
 - nets 5-18

- ports 5-18
- references 5-18
- registers 5-18
- specifying
 - absolute path 5-20
 - relative path 5-18
- design problems
 - commands
 - analyzing B-5
 - resolving B-5
- design reuse
 - partitioning 3-5
- design rule
 - attributes 7-3
- design rule constraints
 - capacitance 7-7
 - defined 4-4
 - fanout load 7-5
 - setting 7-3
 - transition time 7-4
- design rule cost function 8-30
- design rules
 - commands B-2
- Design Vision
 - defined 1-8
- designs
 - analyzing 9-8
 - checking consistency 9-2
 - copying 5-22
 - creating 5-21
 - current 5-5
 - defined 5-3
 - duplicate, checking for 5-12
 - editing 5-38
 - buses 5-38
 - cells 5-38
 - nets 5-35, 5-38
 - ports 5-36, 5-38
 - flat 5-3
 - hierarchical 5-3
 - linking 5-14
 - automatically 5-16

- manually 5-15
- listing
 - details 5-12
 - names 5-11
- listing current 5-13
- parent 5-3
- preserving implementation 8-28
- reading 2-20, 5-8
 - .db format 5-10
 - HDL 5-10
 - HDL (analyze command) 5-11
 - HDL (elaborate command) 5-11
 - netlists 2-20
 - RTL 2-20
- reference, changing 5-16
- removing from memory 5-42
- renaming 5-23
- reporting attributes 8-29
- saving 5-43
 - default behavior 5-43
 - hierarchical 5-44
 - modified 5-44
 - multiple 5-44
 - supported formats 5-43
- setting current 5-13
- translating 5-40
 - with multiple instances 5-6
- DesignWare
 - defined 1-7
- DesignWare library
 - defined 1-7, 4-6
 - file extension 4-7
 - specifying 4-7, 4-10
- DFT Compiler
 - defined 1-7
- directory structure
 - bottom-up compile
 - figure 3-4
 - top-down compile
 - figure 3-3
- disabled timing arc, compared with false path 7-19

- disabling
 - false violation messages 9-14
 - timing paths
 - scan chains 9-14
- disconnect_net command 5-38
- dont_touch attribute 9-25, 9-27, 10-8
 - and dangling logic 9-16, 9-27
 - and timing analysis 8-28
 - reporting, designs 8-29
 - setting 8-28
- drive characteristics
 - removing 6-15
 - setting
 - command to 6-14
 - example of 6-16
- drive resistance, setting 6-15
- drive strength
 - defining 6-13

E

- EDIF format 5-7
- elaborate command 2-20, 5-8
- endpoints, timing exceptions 7-17
- .eqn (Synopsys equation) format 5-7
- examples of
 - ungrouping hierarchy 5-31
- exit command 2-12
- exiting Design Compiler 2-11
- expressions
 - guidelines
 - HDL 3-30

F

- false path
 - compared with disabled timing arc 7-19
 - defined 7-17
 - specifying 7-10, 7-17
- false violation messages, disabling 9-14

- fanout
 - specifying values of 6-18
- fanout load
 - calculating 7-5
 - controlling 7-5
 - cost calculation 8-32
 - defined 7-5
 - removing attribute of 7-7
- fanout load constraints 7-5
- fanout_load attribute 7-5
- feedback loop
 - breaking 9-17
 - identifying 9-17
- file name extensions
 - conventions 3-2
- filename log files 2-14
- filename_log_file variable 2-14
- files
 - command log file 2-13
 - filename log file 2-14
 - script 2-14
- filter command 9-13, 9-14
- find command 9-13, 9-14
- flat design 5-3
- flattening
 - critical path logic 8-49
- flattening design 8-4
- flip-flop
 - defined 3-15
 - inferring 3-16
- formats, design file 5-7
- full_case directive 3-25
- functions
 - guidelines
 - HDL 3-31

G

- gate-level optimization 8-6
- get_attribute command 5-52, 9-24, 9-25

- get_cells command 9-13, 9-14
- get_designs command 9-14
- get_ilm_objects command 10-27
- get_ilms command 10-27
- get_license command 2-16
- glue logic 3-6
- group command 5-25, 8-49
- group_path command 8-34
 - critical_range option 8-46
 - features of 8-45
- grouping
 - adding hierarchy levels 5-24

H

- HDL Compiler
 - defined 1-6
- HDL design, reading 5-10
 - analyze command 5-11
 - elaborate command 5-11
- help
 - accessing 2-12
- hierarchical boundaries
 - wire load model 6-7
- hierarchical cells
 - defined 5-5
 - identifying 9-14
- hierarchical compile
 - See, top-down compile
- hierarchical designs
 - defined 5-3
 - writing 5-44
- hierarchical pin timing constraints, preserving 5-35
- hierarchical pins, preserving timing constraints 5-35
- hierarchy
 - adding levels 5-24
 - changing 5-24
 - changing interactively 5-24
 - displaying 5-24

- merging cells 5-37
- removing levels 5-27, 5-30, 8-57
 - all 5-28
 - one 5-28
- ungrouping automatically 5-31
- high-effort compile 8-51
- hlo_disable_datapath_optimization variable 8-59, 8-73
- hold checks
 - default behavior 7-22
 - overriding default behavior 7-21, 7-23
 - timing arcs and 6-17

I

- ideal clocks 7-12
- identifiers
 - guidelines
 - HDL 3-28
- identifying
 - black box cells 9-14
 - feedback loops 9-17
 - hierarchical cells 9-14
 - unmapped cells 9-13
- if statement 3-23
- ILM (interface logic model) 10-1
- incremental compile 8-52
- infer_multibit compiler directive 3-18
- infer_mux compiler directive 3-15
- inferring registers 3-15
- input arrival time
 - default 7-13
 - removing 7-14
 - reporting 7-14
 - specifying 7-13
- input formats
 - Berkeley Espresso (PLA) 5-8
 - .db 5-7
 - EDIF 5-7
 - .eqn 5-7
 - licensed 5-7

- LSI Logic 5-7
- Mentor Graphics (.mif) 5-7
- supported 5-7
- Synopsys state table (.st) 5-8
- TDL 5-8
- Verilog 5-8
- VHDL 5-8
- Xilinx (.xnf) 5-8
- instances 5-5
 - current 5-5
 - listing 5-17
 - multiple 5-6
 - reporting 5-17
- interblock buffers 9-16
- interface logic model
 - annotated delay, propagating 10-33, 10-34, 10-35
 - back-annotation, applying 10-34
 - back-annotation, using 10-33
 - basic definition 10-3
 - benefits 10-5
 - commands, list and summary of 10-37
 - creating 10-11
 - defined 10-1
 - dont_touch attribute 10-8
 - get_ilm_objects command 10-27
 - get_ilms command 10-27
 - guidelines for creating 10-9
 - instantiating and using in logic synthesis 10-30
 - preserving as subdesign 8-28
 - propagate_ilm command 10-33, 10-34, 10-35
 - read_sdf command 10-35, 10-36
 - report_area command 10-27, 10-29
 - report_design command 10-27, 10-28
 - reporting area information 10-27, 10-29
 - reporting design information 10-27, 10-28
 - reporting design information, using extract_ilm -verbose command 10-27
 - reporting information 10-27

- set_load command 10-5, 10-34, 10-35, 10-36
- smallest model, getting 10-20
- top-down compile 8-9, 8-10
- using, points to consider 10-6
- interfaces
 - dcsh 2-7
 - dctl 2-7
- is_black_box attribute 9-14
- is_hierarchical attribute 9-14
- is_unmapped attribute 9-13

L

- latches
 - defined 3-15
 - inferring 3-15
- leaf cell 5-5
- libraries
 - DesignWare 1-7, 4-6
 - link 4-5
 - list of 6-9
 - list values of 6-9, 6-17
 - listing
 - details 4-11
 - names 4-11
 - main 4-9
 - power consumption 4-5
 - reading 4-11
 - removing from memory 4-16
 - reporting contents 4-12
 - saving 4-16
 - specifying 2-19, 4-7, 4-9
 - objects 4-12
 - symbol 4-6
 - synthetic 3-18
 - target 4-4
 - technology 4-4
 - timing values 4-5
- library attributes C-5

- library cell
 - reference 5-6
 - specifying 5-40
- library cell attributes C-5
- library objects
 - defined 4-12
 - specifying 4-12
- library registers
 - specifying 5-40
- license_users command 2-15
- licenses
 - checking out 2-16
 - design formats
 - special 5-7
 - listing 2-15
 - releasing 2-16
 - using 2-15
 - working with 2-15
- link library
 - file extension 4-7
 - including designs in memory 4-9
 - libraries
 - cell references 4-5
 - specifying 4-7
 - target library and 4-9
- link_force_case variable 5-15
- link_library variable 4-7, 5-14
- list command 6-12
- list -libraries command 4-11
- list_designs command 5-11
- list_duplicate_designs command 5-12
- list_instances command 5-17
- list_libs command 4-11, 6-9
- load_of command 9-24
- log files
 - command log file 2-13
 - filename log file 2-14
- logic-level optimization 8-3
- LSI Logic (NDL) netlist format 5-7

M

- main library 4-9
- man pages
 - accessing 2-12
- max_area attribute 7-25
- max_capacitance attribute 7-8, 9-26
- max_fanout attribute 7-5, 9-26
- max_transition attribute 7-4, 9-26
- maximum delay, calculating cost 8-33
- maximum performance optimization 8-44
- Mentor Graphics
 - input format (.mif) 5-7
 - output format (.neted) 5-7
- messages
 - control echoing to screen 5-55
 - disabling 9-14
- minimization
 - defined 8-41
 - enabling 8-41
- minimum area optimization 8-53
- minimum delay, calculating cost 8-36
- minimum porosity, calculating cost 8-38
- mixed compile strategy 8-19
- model, interface logic, defined 10-1
- modes
 - DB 2-7
 - XG 2-7
- Module Compiler
 - defined 1-7
- modules
 - guidelines
 - HDL 3-33
- multicycle path 7-21
- multiple clock considerations 7-11
- multiple instances of a design
 - resolving 8-20
- multiple instances, resolving
 - compile command automatic unify 8-22
 - compile-once-don't-touch method 8-24
 - ungroup method 8-26

- uniquify method 8-22
- multiplexers
 - inferring 3-15
 - HDL Compiler 3-15

N

- name
 - changing net or port 5-45
- naming conventions
 - file name extensions 3-2
 - library objects 4-12
 - signal name suffixes 3-30
- naming translation 5-45
- net attributes C-5
- net capacitance
 - See, capacitance
- net names
 - changing name rules 5-45
- netlist reader 2-20
- netlists, reading 2-20
- nets 5-7
 - connecting 5-38
 - creating 5-35, 5-38
 - disconnecting 5-38
 - heavily loaded, fixing 8-48
 - reporting 5-18
- networks 5-7

O

- operating conditions
 - defining 6-3
 - list of
 - current design 6-5
 - technology library 6-4
- optimization
 - across hierarchical boundaries 8-56
 - architectural 8-2
 - Boolean 8-54
 - boundary 8-56

- cost function 8-30
- data paths 8-58
- defined 2-2
- gate level 8-6
- gate-level 8-6
- high-speed designs 8-42
- how it works 9-3
- incremental 8-52
- invoking 8-39
- logic-level 8-3
- maximum performance 8-44
- minimum area 8-53
- random logic 8-40
- structured logic 8-42
- trials phase 9-3
- optimization processes 8-2
- output delay
 - default constraint 7-13
 - removing 7-14
 - reporting 7-14
 - specifying 7-13
- output formats
 - Berkeley Espresso (PLA) 5-8
 - .db 5-7
 - EDIF 5-7
 - .eqn 5-7
 - licensed 5-7, 5-43
 - LSI Logic 5-7
 - Mentor Graphics (.neted) 5-7
 - supported 5-7, 5-43
 - Synopsys state table (.st) 5-8
 - TDL 5-8
 - Verilog 5-8
 - VHDL 5-8
 - Xilinx (.xnf) 5-8

P

- partitioning
 - by compile technique 3-9
 - combinational logic 3-5
 - design reuse considerations 3-5

- glue logic 3-6
- merge resources 3-10
- modules by design goals 3-8
- modules with different goals 3-8
- random logic 3-9
- sharable resources 3-10
- structural logic 3-9
- user-defined resources 3-11
- path delay
 - specifying 7-19
- path groups
 - and delay cost 8-35
 - creating 8-34
 - defined 8-34
 - listing 8-34
- paths
 - multicycle 7-21
 - specifying false 7-10
 - using absolute 5-20
 - using relative 5-18
 - using search 5-9
- pin attributes C-6
- pins 5-7
 - library cell, specifying 4-12
 - relationship to ports 5-7
- PLA (Berkeley Espresso) format 5-8
- point-to-point exception
 - See, timing exception
- porosity cost, calculating 8-38
- port
 - names, changing 5-45
- ports 5-7
 - capacitive load on
 - setting 6-17
 - creating 5-36, 5-38
 - deleting 5-38
 - listing
 - output ports 5-18
 - relationship to pins 5-7
 - reporting 5-18
 - setting drive characteristics of 6-14, 6-15

- wire delays, preventing 9-16
- Power Compiler
 - defined 1-7
- preserved timing constraints in design
 - hierarchies 5-37
- preserving subdesigns 8-28
- propagate_ilm command 10-33, 10-34, 10-35

Q

- quit command 2-12
- quitting Design Compiler 2-11

R

- random logic optimization 8-40
- read_file command 2-20, 4-11, 5-8, 5-13
- read_lib command 4-11
- read_sdf command 10-35, 10-36
- read_verilog command 2-20
- read_vhdl command 2-20
- reference attributes C-6
- references
 - changing design 5-16
 - defined 5-6
 - reporting 5-18
 - resolving 5-14
 - automatically 5-16
 - manually 5-15
- register inference
 - D flip-flop 3-16
 - D latch 3-15
 - defined 3-15
 - edge expressions 3-16
- register types
 - mixing 3-16
- registers
 - inferring
 - HDL Compiler 3-15
 - listing 5-18
- remove_attribute command 7-5, 7-7, 7-8

- remove_bus command 5-38
- remove_cell command 5-38
- remove_clock command 7-12
- remove_constraint command 7-26
- remove_design command 4-16, 5-42
- remove_input_delay command 7-14
- remove_license command 2-16
- remove_multibit command 3-18
- remove_net command 5-38
- remove_output_delay command 7-14
- remove_port command 5-38
- remove_wire_load_model command 6-13
- removing levels of hierarchy 5-27
- rename_design command 5-23
- report_area command 10-27, 10-29
- report_attribute command 5-52
- report_auto_ungroup 5-32
- report_cell command 9-25
- report_clock command 5-18
 - purpose 7-12
 - skew option 7-13
- report_compile_options command 8-5
- report_constraint command 9-18, 9-20
 - all_violators option
 - report violations 9-20
 - verbose option 9-19
- report_delay_calculation command 9-11
- report_design command 6-5, 8-29, 10-27, 10-28
- report_hierarchy command 5-24
- report_lib command 4-12, 6-4, 6-9, 6-17
- report_net command 5-18
- report_path_group command 8-34
- report_port command 5-18, 7-14
- report_reference command 5-18
- report_resources command 8-70
- report_timing command
 - feedback loops 9-17
 - wire load information 6-13

- report_timing_requirements command
 - delay requirements 7-16
 - ignored option 7-17
 - timing exceptions 7-17
- reports
 - analyzing design 9-8
 - analyzing timing 9-9
 - check_design command 9-2
 - clock definition 7-12
 - delay calculation 9-11
 - library contents 4-12
 - operating condition 6-4
 - operating conditions 6-5
 - report_hierarchy command 5-24
 - script file A-26, A-37
 - timing exceptions 7-17
 - ignored 7-17
 - timing path 9-11
 - wire load model
 - example 6-10
- reset_path command 7-18, 7-19, 7-23
- resistance
 - output driver
 - defining 6-13
 - See also, drive characteristics
- resolving multiple instances of a design 8-20
- resource allocation
 - area driven 8-55
 - timing driven 8-55
- resources
 - shareable 3-10
 - user-defined
 - partitioning 3-11
- routability cost
 - See, porosity cost
- RTL, reading 2-20

S

- script files 2-14
 - adding comments 2-15
 - compile A-16

- executing 2-15
 - generating 5-53
 - report A-26, A-37
 - return values 2-15
- search path
 - for design files 5-9
 - for libraries 4-7
- search_path variable 4-7, 5-9
- semiconductor vendor, selecting 4-3
- sequential device, initialize or control state 3-16
- set_clock_latency command
 - setting margin of error 7-13
- set_clock_uncertainty command 7-10
- set_cost_priority command 8-31
- set_critical_range command 8-46
- set_disable_timing command 9-14
- set_dont_touch command 8-24, 8-28
- set_drive command 6-14, 6-15, 6-16
- set_driving_cell command 6-13, 6-14, 6-16
- set_false_path command 7-17
 - undoing 7-18
 - uses for 7-17
- set_fanout_load command 6-18, 7-6
- set_flatten command 8-4
 - minimize option 8-41
 - phase option 8-41
- set_input_delay command 7-10, 7-13
- set_input_transition command 6-14
- set_load command 6-17, 10-5, 10-34, 10-35, 10-36
- set_max_area command 7-25
- set_max_delay command
 - for combinational paths 7-16
 - for timing exceptions 7-19
 - reset 7-19
- set_max_fanout command 7-6
- set_max_transition command 7-4
- set_min_delay command
 - for combinational paths 7-16
 - for timing exceptions 7-19
 - reset 7-19
- set_multicycle_path command 7-21
 - default behavior 7-21
 - reset 7-23
- set_output_delay command 7-10, 7-13
- set_resistance command 9-16
- set_structure command 8-3, 8-54
- set_ultra_optimization command 8-73
- set_ungroup command 5-30, 8-57
- set_wire_load command 6-7, 6-12
- setup checks
 - default behavior 7-22
 - overriding default behavior 7-21, 7-22
 - timing arcs and 6-17
- setup files
 - design example A-12
 - .synopsys_dc.setup file 2-8
- sh_command_log_file variable 2-14
- signal_type attribute 9-15
- signals, edge detection 3-16
- slack
 - critical negative 8-35
 - total negative 8-35
 - worst negative 8-33
- specifying
 - clock
 - network delay 7-12
 - period 7-11
 - waveform 7-11
 - libraries
 - DesignWare 4-10
 - link 4-7
 - symbol 4-7
 - target 4-7
 - library objects 4-12
 - maximum transition time 7-4
 - timing exceptions
 - false path 7-17
 - multicycle path 7-21
 - path delay 7-19

- timing requirements
 - combinational paths 7-16
 - input ports 7-13
 - output ports 7-13
 - wire load mode 6-12
 - wire load model 6-12
- .st (Synopsys state table) format 5-8
- starting dc_shell 2-10
- startpoints, timing exceptions 7-17
- state machine design 3-21
- statements
 - 'define 3-26
 - case 3-24
 - constant 3-26
 - if 3-23
- structured logic optimization 8-42
- structuring design
 - optimization 8-3
- subdesigns 5-3
 - preserving 8-28
- symbol library
 - defined 4-6
 - file extension 4-7
 - search path for 4-7
 - specifying 4-7
- symbol_library variable 4-7
- sync_set_reset directive 3-17
- synchronous designs
 - clock period 7-11
- Synopsys state table (.st) format 5-8
- .synopsys_dc.setup file 2-8
 - sample 2-9
- synthesis
 - defined 2-2
 - using interface logic models 10-30
- synthesis design flow
 - figure 2-18
- synthetic libraries 3-18
- synthetic_library variable 4-7

T

- target library
 - definition 4-4
 - file extension 4-7
 - link library and 4-9
 - specifying 4-7
- target_library variable 4-7
- TDL (Tegas Design Language) format 5-8
- technology library
 - creating 4-4
 - definition 4-4
 - required format 4-4
 - search path for 4-7
- timing
 - analyzing 9-9
 - reports 7-16
- timing arcs
 - hold checks and 6-17
 - setup checks and 6-17
- timing constraints, commands to set 7-9
- timing exception
 - commands
 - listing 7-17
 - order of precedence 7-23, 7-24
 - defined 7-16
 - ignored, list of 7-17
 - reporting 7-17
 - valid endpoints 7-17
 - valid startpoints 7-17
- timing path, report 9-11
- timing values
 - link libraries 4-5
- timing violations
 - correcting 9-25
 - scan chain 9-14
- top-down compile 8-9
 - advantages 8-10
 - directory structure
 - figure 3-3
- total negative slack, defined 8-35

- transition time
 - cost calculation 8-31
 - defined 7-4
 - setting 7-4
 - specifying
 - maximum 7-4
- translate command 5-40, 5-41
- translating designs
 - procedure for 5-40
 - restrictions 5-41

U

- ungroup command 5-28, 8-26
- ungroup design
 - compile option 5-30, 8-57
- ungroup hierarchy
 - examples 5-31
- ungroup_preserve_constraints variable 5-35
- ungrouping
 - automatically 5-31, 8-51
 - cell count-based 5-32
 - delay-based 5-33
 - removing hierarchy levels 5-27, 5-30, 8-57
- uniquify command 8-22
- uniquify method 8-21

V

- variables
 - auto_ungroup_preserve_constraints 5-34
 - command_log_file 2-14
 - compile_assume_fully_decoded_three_state_busses 5-42
 - compile_auto_ungroup_area_num_cells 5-32, 5-34
 - compile_auto_ungroup_delay_num_cells 5-33, 5-34, 8-43
 - compile_automongroup_override_wlm 5-34
 - compile_fix_cell_degradation 8-33
 - compile_log_format 9-4
 - compile_new_boolean_structure 8-54

- compile_sequential_area_recovery 8-54
- current_design 5-13
- current_instance 5-19
- filename_log_file 2-14
- hlo_disable_datapath_optimization 8-59, 8-73
- link_force_case 5-15
- link_library 4-7, 5-14
- search_path 4-7, 5-9
- sh_command_log_file variable 2-14
- symbol_library 4-7
- synthetic_library 4-7
- target_library 4-7
- ungroup_preserve_constraints 5-35

Verilog

- expressions 3-30
- format 5-8
- functions 3-31
- identifiers 3-28
- modules 3-33

VHDL

- expressions 3-30
- format 5-8
- functions 3-31
- identifiers 3-28
- modules 3-33

virtual clock

- creating 7-12
- defined 7-12

W

- wire delays, on ports 9-16
- wire load
 - defining 6-5
- wire load mode
 - default 6-12
 - reporting 6-13
 - specifying 6-12
- wire load model
 - automatic selection
 - described 6-11

- disabling 6-12
- choosing 6-12
- default 6-11
- hierarchical boundaries 6-7
- list of
 - technology libraries 6-9
- removing 6-13
- report example 6-10
- reporting 6-13
- specifying 6-12

- wire_load_selection library function 6-11
- worst negative slack, defined 8-33
- write command 5-43
- write_lib command 4-16
- write_script command 5-53

X

- XG mode 2-7
- Xilinx (.xnf) format 5-8