Design Compiler®
User Guide

Version D-2010.03-SP2, June 2010

SYNOPSYS®

Copyright Notice and Proprietary Information

Copyright © 2010 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, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, Certify, CHIPit, Design Compiler, DesignWare, Formality, HAPS, HDL Analyst, HSIM, HSPICE, Identify, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Physical Compiler, PrimeTime, SCOPE, Simply Better Results, SiVL, SNUG, SolvNet, Syndicated, Synplicity, Synplify, Synplify Pro, Synthesis Constraints Optimization Environment, TetraMAX, the Synplicity logo, UMRBus, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, BEST, Columbia, Columbia-CE, Confirma, Cosmos, CosmosLE, CosmosScope, CRITIC, CustomExplorer, CustomSim, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, DesignPower, DFTMAX, Direct Silicon Access, Discovery, Eclypse, Encore, EPIC, Galaxy, Galaxy Custom Designer, HANEX, HapsTrak, HDL Compiler, Hercules, Hierarchical Optimization Technology, High-performance ASIC Prototyping System, HSIM^{plus}, i-Virtual Stepper, IICE, in-Sync, iN-Tandem, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Magellan, Mars, Mars-Rail, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, MultiPoint, Physical Analyst, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Saturn, Scirocco, Scirocco-i, Star-RCXT, Star-SimXT, StarRC, System Compiler, System Designer, Taurus, TotalRecall, TSUPREM-4, VCS Express, VCSi, VHDL Compiler, 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.
Saber is a registered trademark of SabreMark Limited Partnership and is used under license.
All other product or company names may be trademarks of their respective owners.

Contents

What's New in This Release xviii

About This Manual xviii

Customer Support. xxi

1. Introduction to Design Compiler

Design Compiler and the Design Flow 1-2

Design Compiler Family 1-3
- DC Expert 1-4
- DC Ultra 1-4
- HDL Compiler Tools 1-5
- DesignWare Library 1-5
- DFT Compiler 1-5
- Power Compiler 1-5
- Design Vision 1-5

2. Design Compiler Basics

The High-Level Design Flow 2-2

Running Design Compiler 2-4
- Design Compiler Interfaces 2-5
- Setup Files 2-5
- Starting Design Compiler 2-7
- Exiting Design Compiler 2-8
- Opening and Closing the GUI in dc_shell 2-8

Getting Command Help 2-9
Using Command Log Files 2-10
Using the Filename Log File 2-10
Using Script Files. 2-10

Support for Multicore Technology 2-11

Support for Multicorner-Multimode Designs 2-12

Working With Licenses 2-12
Listing the Licenses in Use 2-12
Getting Licenses 2-13
Enabling License Queuing. 2-13
Releasing Licenses 2-14

Following the Basic Synthesis Flow 2-15

A Design Compiler Session Example 2-20

3. Preparing Design Files for Synthesis

Managing the Design Data 3-2
Controlling the Design Data. 3-2
Organizing the Design Data. 3-2

Partitioning for Synthesis 3-4
Partitioning for Design Reuse 3-4
Keeping Related Combinational Logic Together 3-5
Registering Block Outputs 3-6
Partitioning by Design Goal 3-7
Partitioning by Compile Technique. 3-7
Keeping Sharable Resources Together 3-8
Keeping User-Defined Resources With the Logic They Drive 3-8
Isolating Special Functions 3-9

HDL Coding for Synthesis. 3-10
Writing Technology-Independent HDL 3-11
Inferring Components 3-11
Using Synthetic Libraries 3-13
Designing State Machines 3-15
Using HDL Constructs 3-16
General HDL Constructs. 3-17
Using Verilog Macro Definitions 3-20

Using VHDL Port Definitions. 3-20
Writing Effective Code 3-21
Guidelines for Identifiers 3-21
Guidelines for Expressions 3-22
Guidelines for Functions 3-23
Guidelines for Modules 3-24

4. Working With Libraries

Selecting a Semiconductor Vendor. 4-2

Understanding the Library Requirements 4-2
Technology Libraries 4-2
Symbol Libraries 4-4
DesignWare Libraries 4-4

Specifying Libraries 4-5
Specifying Technology Libraries 4-5
Target Library 4-5
Link Library 4-5
Specifying DesignWare Libraries. 4-7
Specifying a Library Search Path 4-8

Loading Libraries 4-8

Listing Libraries. 4-8

Reporting Library Contents. 4-9

Specifying Library Objects 4-9

Directing Library Cell Usage 4-10
Excluding Cells From the Target Library 4-10
Specifying Cell Preferences. 4-10

Library-Aware Mapping and Synthesis 4-11
Generating the ALIB file. 4-12
Using the ALIB library 4-12

Removing Libraries From Memory 4-13

Saving Libraries 4-13

5. Working With Designs in Memory

Design Terminology 5-2

About Designs 5-2
Flat Designs 5-2
Hierarchical Designs. 5-2
Design Objects 5-2
Relationship Between Designs, Instances, and References 5-4
Reporting References. 5-5
Using Reference Objects 5-5
Reading Designs 5-6
Commands for Reading Design Files 5-6
Using the analyze and elaborate Commands. 5-6
Using the read_file Command 5-8
Reading HDL Designs 5-10
Reading .ddc Files 5-11
Reading .db Files. 5-11
Listing Designs in Memory 5-11
Setting the Current Design 5-12
Using the current_design Command 5-13
Linking Designs 5-13
Locating Designs by Using a Search Path. 5-15
Changing Design References 5-15
Listing Design Objects 5-16
Specifying Design Objects 5-17
Using a Relative Path 5-17
Using an Absolute Path 5-18
Creating Designs 5-19
Copying Designs. 5-19
Renaming Designs 5-20
Changing the Design Hierarchy 5-21
Adding Levels of Hierarchy 5-22
Grouping Cells Into Subdesigns 5-22
Removing Levels of Hierarchy 5-25
Ungrouping Hierarchies Explicitly During Optimization 5-27
Ungrouping Hierarchies Automatically During Optimization 5-28
Merging Cells From Different Subdesigns 5-30

Editing Designs 5-31

Translating Designs From One Technology to Another 5-33
Procedure to Translate Designs 5-33
Restrictions on Translating Between Technologies 5-34

Removing Designs From Memory 5-34

Saving Designs 5-35
Commands to Save Design Files 5-35
Using the write Command 5-35
Using the write_milkyway Command 5-36
Saving Designs in .ddc Format 5-36
Ensuring Name Consistency Between the Design Database and the Netlist . . 5-37
Naming Rules Section of the .synopsys_dc.setup File 5-37
Using the define_name_rules -map Command 5-37
Resolving Naming Problems in the Flow 5-38

Working With Attributes 5-40
Setting Attribute Values 5-41
Using an Attribute-Specific Command 5-41
Using the set_attribute Command 5-41
Viewing Attribute Values 5-42
Saving Attribute Values 5-42
Defining Attributes 5-43
Removing Attributes 5-43
The Object Search Order 5-43

6. Defining the Design Environment

Defining the Operating Conditions 6-3
Determining Available Operating Condition Options 6-3
Specifying Operating Conditions 6-4

Defining Wire Load Models 6-4
Hierarchical Wire Load Models 6-5
Determining Available Wire Load Models 6-7
Specifying Wire Load Models and Modes 6-8

Modeling the System Interface 6-9
Defining Drive Characteristics for Input Ports 6-10
The set_driving_cell Command 6-10

The set_drive and set_input_transition Commands 6-11
Defining Loads on Input and Output Ports. 6-12
Defining Fanout Loads on Output Ports. 6-13

Setting Logic Constraints on Ports 6-13
Defining Ports as Logically Equivalent. 6-14
Defining Logically Opposite Input Ports. 6-14
Allowing Assignment of Any Signal to an Input 6-15
Specifying Input Ports Always One or Zero 6-15
Tying Input Ports to Logic 1 6-16
Tying Input Ports to Logic 0 6-16
Specifying Unconnected Output Ports. 6-17

Specifying Power Intent 6-17
Power Intent Concepts. 6-18
UPF Commands in Synopsys Tools 6-19
upf_version 6-21
set_scope 6-21
load_upf 6-21
save_upf 6-22

Support for Multicorner-Multimode Designs 6-22

7. Defining Design Constraints

Design Compiler Constraint Types 7-2

Design Rule Constraints 7-3
Design Rule Cost Function 7-4
Maximum Transition Time 7-4
Defining Maximum Transition Time. 7-5
Specifying Clock-Based Maximum Transition 7-5
Maximum Fanout 7-6
Maximum Fanout Calculation Example 7-6
Defining Maximum Fanout 7-7
Defining Expected Fanout for Output Ports 7-8
Maximum Capacitance 7-8
Defining Maximum Capacitance. 7-9
Specifying Frequency-Based Maximum Capacitance 7-9
Minimum Capacitance 7-10
Defining Minimum Capacitance 7-10
Cell Degradation 7-11

Connection Class. 7-12
Managing Design Rule Constraint Priorities 7-12
Precedence of Design Rule Constraints. 7-12
Design Rule Scenarios 7-13
Disabling Design Rule Fixing on Special Nets. 7-14
Summary of Design Rule Commands and Objects 7-14
Optimization Constraints. 7-15
Optimization Cost Function 7-15
Timing Constraints. 7-15
Maximum Delay 7-16
Minimum Delay 7-17
Maximum Area. 7-18
Cost Calculation 7-18
Defining Maximum Area 7-19
Managing Constraint Priorities 7-19
Reporting Constraints. 7-21
Propagating Constraints in Hierarchical Designs 7-22
Characterizing Subdesigns 7-22
Using the characterize Command 7-22
Removing Previous Annotations. 7-24
Optimizing Bottom Up Versus Optimizing Top Down 7-24
Deriving the Boundary Conditions 7-24
Limitations of the characterize Command 7-25
characterize Command Calculations 7-26
Characterizing Subdesign Port Signal Interfaces 7-29
Characterizing Subdesign Constraints 7-32
Characterizing Subdesign Logical Port Connections 7-33
Characterizing Multiple Instances. 7-33
Characterizing Designs With Timing Exceptions 7-34
Propagating Constraints up the Hierarchy. 7-36
Methodology for Propagating Constraints Upward 7-37
Handling of Conflicts Between Designs 7-37

8. Optimizing the Design

The Optimization Process. 8-2
Architectural Optimization 8-2
Logic-Level Optimization 8-3

Gate-Level Optimization 8-3
Selecting and Using a Compile Strategy 8-4
Top-Down Compile 8-5
Bottom-Up Compile 8-7
Mixed Compile Strategy 8-11
Resolving Multiple Instances of a Design Reference 8-11
Uniquify Method 8-13
Compile-Once-Don't-Touch Method 8-14
Ungroup Method 8-16
Preserving Subdesigns 8-17
Understanding the Compile Cost Function 8-18
Performing Design Exploration 8-19
Performing Design Implementation 8-20
Optimizing High-Performance Designs 8-20
Optimizing for Maximum Performance 8-21
Creating Path Groups 8-21
Fixing Heavily Loaded Nets 8-23
Automatically Ungrouping Hierarchies on the Critical Path 8-24
Performing a High-Effort Compile 8-24
Performing a High-Effort Incremental Compile 8-25
Optimizing for Minimum Area 8-25
Disabling Total Negative Slack Optimization 8-26
Optimizing Across Hierarchical Boundaries 8-26
Optimizing Data Paths 8-27

9. Using Interface Logic Models

Overview of Interface Logic Models (ILMs) 9-2
General Guidelines for Creating ILMs 9-4
Controlling the Logic Included in ILMs 9-5
Basic ILM Logic Content 9-5
Controlling the Fanins and Fanouts of Chip-Level Networks 9-6
Controlling the Number of Latch Levels 9-7
Controlling Side-Load Cells 9-8
Methodology for Creating and Saving ILMs 9-9

Instantiating and Using ILMs in the Top-Level Design 9-12
Using the create_ilm Command Options 9-15
Using ILMs with Multicorner-Multimode Designs 9-16
Reporting Information About ILMs 9-16
Summary of Commands for ILMs 9-19

10. Using Design Compiler Topographical Technology

Overview of Topographical Technology. 10-3
Starting Design Compiler Topographical Mode 10-7
Inputs and Outputs in Design Compiler Topographical Mode. 10-7
Specifying Libraries 10-9
Specifying Logical Libraries. 10-9
Specifying Physical Libraries 10-10
Verifying Library Consistency 10-11
Using TLUPlus for RC Estimation 10-13
Support for Black Boxes 10-14
Using Floorplan Physical Constraints 10-15
Physical Constraints Overview 10-15
Extracting Physical Constraints From IC Compiler Using the write_def Command 10-16
Importing Physical Constraints From an IC Compiler DEF Formatted File 10-17
Importing a DEF Floorplan Overview 10-17
Imported DEF Constraints 10-18
Importing Incremental DEF Information Using the extract_physical_constraints Command 10-24
Matching Names of Macros and Ports 10-25
Exporting Physical Constraints From IC Compiler Using the write_floorplan Command 10-26
Importing Physical Constraints From an IC Compiler write_floorplan Formatted File 10-26
Incremental Floorplan Modifications Using the read_floorplan Command 10-28
Matching Names of Macros and Ports 10-28
Manually Defining Physical Constraints. 10-29
Defining Physical Constraints Overview 10-30
Defining the Die Area With the create_die_area Command. 10-31

Defining the Core Placement Area With the create_site_rows Command 10-33
Defining Placement Area With the set_aspect_ratio and set_utilization Commands 10-33
Defining Port Locations. 10-34
Defining Macro Location and Orientation 10-36
Defining Placement Blockage. 10-37
Defining Voltage Area 10-37
Defining Placement Bounds 10-38
Creating Wiring Keepouts. 10-41
Creating Preroutes 10-41
Specifying Relative Placement 10-44
Benefits of Relative Placement. 10-45
Methodology for the Relative Placement Flow 10-46
Summary of Relative Placement Commands. 10-48
Creating Relative Placement Using Compiler Directives 10-49
Creating Relative Placement Groups 10-49
Anchoring Relative Placement Groups. 10-51
Applying Compression to Relative Placement Groups. 10-51
Specifying Alignment 10-52
Adding Objects to a Group 10-54
Querying Relative Placement Groups 10-61
Checking Relative Placement Constraints 10-62
Saving Relative Placement Information 10-63
Removing Relative Placement Group Attributes. 10-64
Sample Script for a Relative Placement Flow. 10-65
Placement Options: Magnet Placement 10-66
Resetting Physical Constraints 10-67
Saving Physical Constraints Using the write_floorplan Command 10-67
Reporting Physical Constraints 10-68
Performing Automatic High-Fanout Synthesis 10-68
Test Synthesis in Topographical Mode 10-68
Using Power Compiler in Topographical Mode 10-69
Multivoltage Designs. 10-71
Compile Flows in Topographical Mode 10-72
Performing an Incremental Compile 10-73
Performing a Bottom-up or Hierarchical Compile 10-74
Overview of Bottom-Up Compile 10-75

Compiling the Subblock 10-77
Compiling the Design at the Top Level 10-79
Performing Top-Level Design Stitching 10-81
Steps in the Top-Level Design Stitching Flow 10-82

Supported Commands, Command Options, and Variables 10-84

Using the Design Compiler Graphical Tool 10-84
Reducing Routing Congestion 10-85
Routing Congestion Overview 10-85
Viewing Congestion With the Design Vision Layout Window 10-87
Routing Congestion Reduction Flow 10-87
Predicting, Analyzing, and Minimizing Routing Congestion 10-88
Specifying Congestion Optimization Options 10-90
Reporting Congestion 10-91
Summary of Routing Congestion Commands 10-92
Improving Area Correlation and Runtime with Synopsys Physical Guidance (SPG) 10-92
Using Physical Guidance in Design Compiler (compile_ultra -spg) 10-93
Using Physical Guidance in IC Compiler (place_opt -spg) 10-93
Supported Flows 10-93
Reporting Physical Guidance Information 10-93
Physical Guidance Limitations 10-93
Creating and Modifying Floorplans Using Floorplan Exploration 10-94
Floorplan Exploration Overview 10-94
Enabling Floorplan Exploration 10-95
Using the Floorplan Exploration GUI 10-97
Creating and Editing Floorplans 10-98
Saving the Floorplan or Discarding Updates 10-99
Saving the Floorplan into a Tcl Script File or DEF File 10-99
Exiting the Session 10-100
Incremental or Full Synthesis after Floorplan Changes 10-101
Floorplan Exploration Limitations 10-101

Optimizing Multicorner-Multimode Designs in Design Compiler Graphical 10-102
Multicorner-Multimode Concepts 10-103
Multicorner-Multimode Feature Support 10-104
Unsupported Features 10-104
Concurrent Multicorner-Multimode Optimization and Timing Analysis 10-105
Basic Multicorner-Multimode Flow 10-105
Setting Up the Design for a Multicorner-Multimode Flow 10-107
Specifying TLUPlus Files 10-107

Specifying Operating Conditions 10-108
Specifying Constraints 10-108
Handling Libraries in the Multicorner-Multimode Flow. 10-109
Using Link Libraries That Have the Same PVT Nominal Values 10-109
Using Unique PVT Names to Prevent Linking Problems 10-111
Unsupported k-factors 10-113
Automatic Detection of Driving Cell Library 10-113
Defining Minimum Libraries 10-113
Scenario Management Commands 10-114
Creating Scenarios 10-115
Defining Active Scenarios. 10-116
Scenario Reduction 10-116
Specifying Scenario Options 10-116
Removing Scenarios. 10-117
Power Optimization Techniques. 10-117
Optimizing for Leakage Power 10-118
Optimizing for Dynamic Power 10-119
Reporting Commands 10-120
report_scenario Command 10-121
report_scenario_options Command 10-121
Reporting Commands That Support the -scenario Option 10-122
Commands That Report the Current Scenario 10-123
Reporting Examples 10-124
Supported SDC Commands 10-129
Multicorner-Multimode Script Example 10-129
Using ILMs in Multicorner-Multimode Designs 10-131
Methodology for Using ILMs With Scenarios at the Top Level 10-131

11. Using a Milkyway Database

Licensing and Required Files 11-2
Invoking the Milkyway Tool 11-2
About the Milkyway Database. 11-2
Guidelines for Using the Milkyway Databases 11-3
Preparing to Use the Milkyway Database 11-4
Writing the Milkyway Database. 11-5
Important Points About the write_milkyway Command 11-5
Results of Running the write_milkyway Command 11-6

Limitations When Writing Milkyway Format 11-6
Script to Set Up and Write a Milkyway Database 11-7
Maintaining the Milkyway Design Library 11-7
Setting the Milkyway Design Library for Writing an Existing Milkyway Database . . . 11-7

12. Analyzing and Resolving Design Problems

Instantiating RTL PG Pins in a Non-UPF Mode 12-2
Resolving Bus Versus Bit-Blasted Mismatches Between the RTL and Macros 12-3
Fixing Errors Caused by New Unsupported Technology File Attributes 12-3
Using Register Replication to Solve Timing QoR,
Congestion, and Fanout Problems 12-4
Comparing Design Compiler Topographical and IC Compiler Environments 12-4
Assessing Design and Constraint Feasibility in Mapped Designs 12-5
Checking for Design Consistency 12-6
Analyzing Your Design During Optimization 12-7
Customizing the Compile Log 12-8
Saving Intermediate Design Databases 12-9
Analyzing Design Problems 12-10
Analyzing Area 12-10
Analyzing Timing 12-11
Resolving Specific Problems 12-13
Analyzing Cell Delays 12-13
Finding Unmapped Cells 12-14
Finding Black Box Cells 12-15
Finding Hierarchical Cells 12-15
Disabling Reporting of Scan Chain Violations 12-15
Insulating Interblock Loading 12-16
Preserving Dangling Logic. 12-16
Preventing Wire Delays on Ports 12-16
Breaking a Feedback Loop 12-16
Analyzing Buffer Problems. 12-17
Understanding Buffer Insertion. 12-17
Correcting for Missing Buffers. 12-21

Correcting for Extra Buffers 12-24
Correcting for Hanging Buffers 12-24
Correcting Modified Buffer Networks 12-24

Appendix A. Design Example

Design Description A-2

Setup File A-9

Default Constraints File. A-10

Read Script A-10

Compile Scripts. A-11

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-4

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

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

To see the *Design Compiler Release Notes,*

1. Go to the release notes page on SolvNet located at the following address:

   https://solvnet.synopsys.com/ReleaseNotes

   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.
2. Select Design Compiler, then select a release in the list that appears at the bottom.

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 B-2008.09 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.

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 Documentation on the Web, which is available through SolvNet at the following address:

https://solvnet.synopsys.com/DocsOnWeb

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
- 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.
`Courier italic`	Indicates a user-defined value in Synopsys syntax, such as *`object_name`*. (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 *`pin1`* `[`*`pin2 ... pinN`*`]`
|	Indicates a choice among alternatives, such as `low | medium | high` (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 `set_annotated_delay`
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

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

Accessing SolvNet

SolvNet includes 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, go to the SolvNet Web page at the following address:

https://solvnet.synopsys.com

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

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

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), and 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/GlobalSupportCenters/Pages
- 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/GlobalSupportCenters/Pages

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.

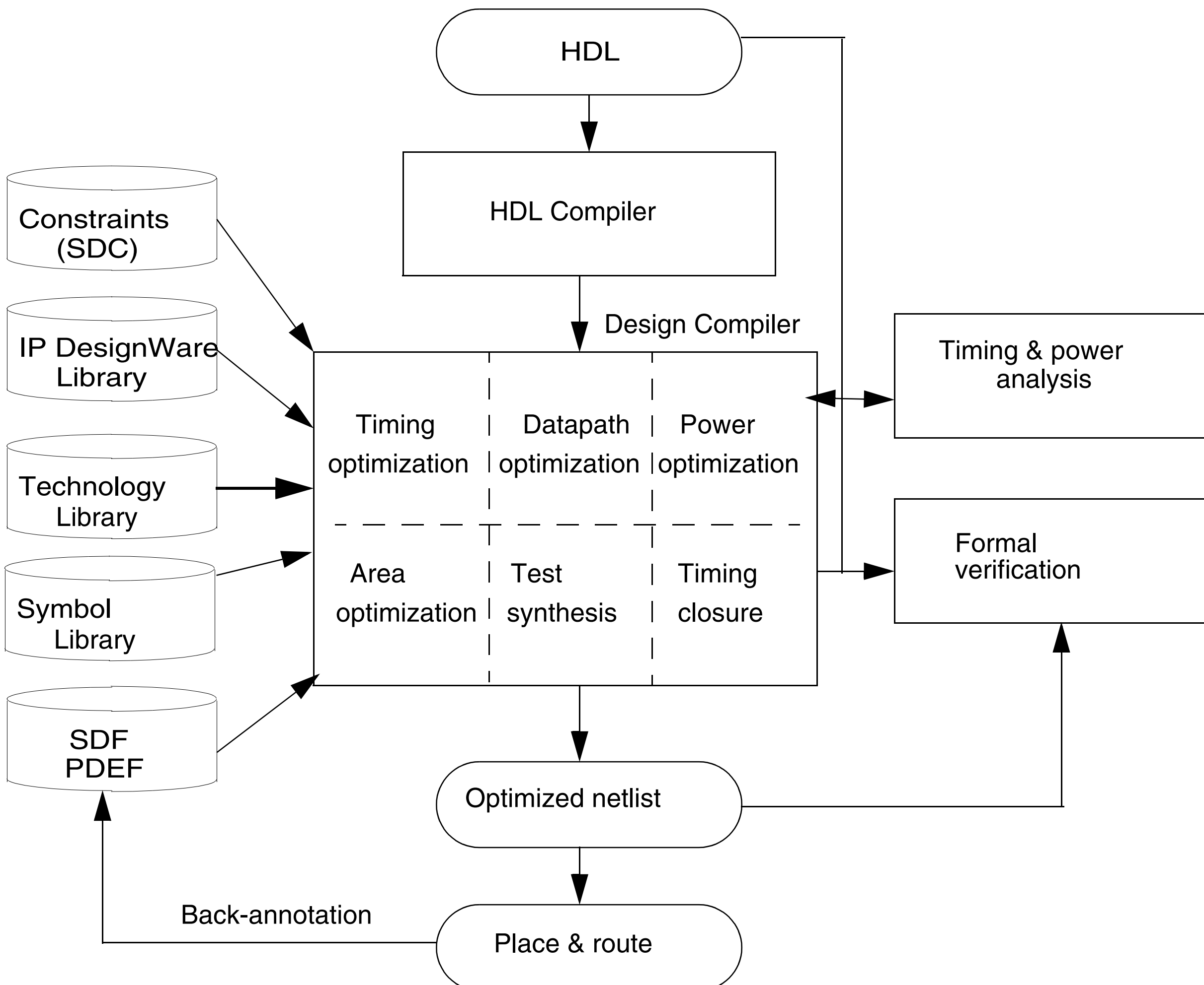
This chapter includes the following sections:

- 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 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
- 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
- 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.

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 tool is a graphical user interface (GUI) to the Synopsys synthesis environment and an analysis tool for viewing and analyzing 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*.

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
- Support for Multicore Technology
- Support for Multicorner-Multimode Designs
- Working With Licenses
- 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.

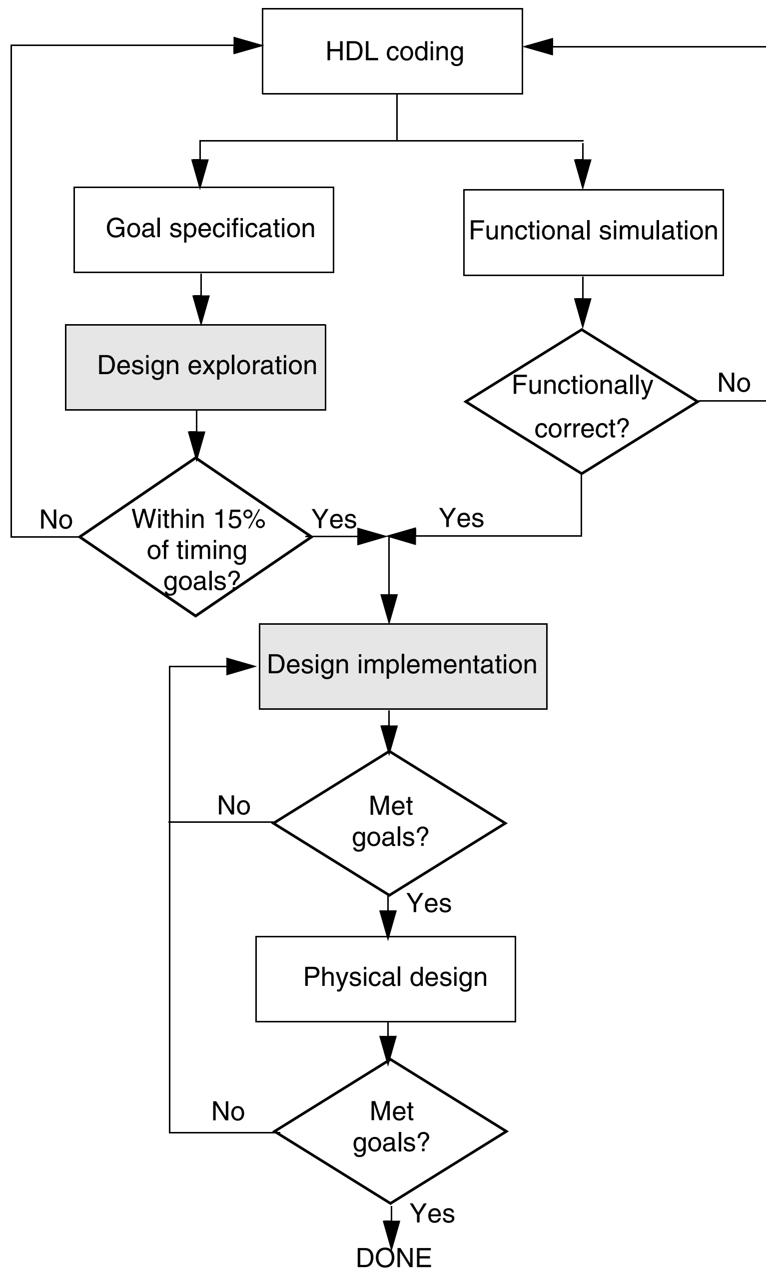
Compile is the Design Compiler command and process that executes the optimization step. After you read in the design and perform other necessary tasks, you invoke the `compile_ultra` command or `compile` command to generate a gate-level netlist for the design.

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 Interfaces
- Setup Files
- Starting Design Compiler
- Exiting Design Compiler
- Getting Command Help
- Using Command Log Files

- Using Script Files
- Working with Licenses

Design Compiler Interfaces

Design Compiler offers two interfaces for synthesis and timing analysis: the dc_shell command-line interface (or shell) and the Design Vision graphical user interface (GUI). The dc_shell command-line interface is a text-only environment in which you enter commands at the command-line prompt. Design Vision is the GUI for the Synopsys synthesis environment; use it for visualizing design data and analysis results. For information on Design Vision, see the *Design Vision User Guide*.

You can interact with the Design Compiler shell by using dctcl, which is based on the tool command language (Tcl) and includes certain command extensions needed to implement specific Design Compiler functionality.

The dctcl command language provides 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
- By typing single commands interactively on the console command line in the Design Vision window

  You can use this approach to supplement the subset of Design Compiler commands available through the menu interface. For more information on Design Vision, see the *Design Vision User Guide* and Design Vision online Help.

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. Note: $SYNOPSYS is the path to the Design Compiler installation directory.
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

To start Design Compiler, enter dc_shell on the command line.

The resulting command prompt is

dc_shell>

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
- `-no_init` to specify that dc_shell is not to execute any .synopsys_dc.setup startup files. Use this option only when you want to include a command log or other script file in order to reproduce a previous dc_shell session.
- `-no_home_init` to specify that dc_shell is not to execute any home .synopsys_dc.setup startup files.
- `-no_local_init` to specify that dc_shell is not to execute any local .synopsys_dc.setup startup files.

For a detailed list of options, see the *Design Compiler Command Line Interface Guide* and the man page 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.

Note:
It is important that you specify the absolute path when entering dc_shell on the command line. If you use a relative path (..\), Design Compiler cannot access the libraries that are located in the root directory.

The following example shows the correct way to indicate the Synopsys root containing the Design Compiler installation.

```
/tools/synopsys/2009.06/bin/dc_shell
```

The following example shows the incorrect way to indicate the Synopsys root containing the Design Compiler installation.

```
../../2009.06/bin/dc_shell
```

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. However, if you change the name of the `sh_command_log_file` 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 -format ddc -hierarchy -output my_design.ddc
```

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.

Opening and Closing the GUI in dc_shell

You can open or close the GUI at any time during the session. If you start dc_shell without the GUI, you can open the GUI by entering the `gui_start` command at the dc_shell prompt.

To open the GUI,

- Enter the `gui_start` command.

```
dc_shell> gui_start
```

The `-gui` option opens the GUI automatically when starting dc_shell.

```
% dc_shell -gui
```

To close the GUI, do one of the following:

- Choose File > Close GUI.
- Enter the gui_stop command.

```
dc_shell> gui_stop
```

When you are ready to use the GUI again, enter the gui_start command on the command line interface.

When you start dc_shell with or without the -gui option, it loads GUI preferences from the .synopsys_dv_prefs.tcl file in your home directory. When you open the GUI, dc_shell reads and executes the .synopsys_dv_gui.tcl setup files, opens the Design Vision window, and displays the dc_shell command-line prompt in the console.

To save GUI window images, use the gui_write_window_image command. You can specify the format of the image as BMP, JPG, XPM, or PNG. The default format is PNG. You can also save window images in batch mode from a script. For details, see the Design Vision Online Help and the gui_write_window_image man page.

For information about using the GUI, see the *Design Vision User Guide*.

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 the command:

```
dc_shell> help
```

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, 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 `sh_command_log_file` variable 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 the 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 dctcl, a “#” at the beginning of a line denotes a comment. For example,

```
# This is a comment
```

To execute a script file, use the `source` command.

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

Support for Multicore Technology

The multicore technology in Design Compiler allows you to use multiple cores to improve the tool runtime. During synthesis, multicore functionality can divide large optimization tasks into smaller tasks for processing on multiple cores.

Multicore technology is only supported in DC Ultra. It is not supported in DC Expert.

The following terms are used in multicore technology: threading, multithreading, and multiple processes.

- Threading: Allows an executing program to split itself into two or more simultaneously running tasks. Threads share the state information and address space of a single process, while processes are independent. Switching between threads in a single processor is easier than switching between processes.
- Multithreading: An execution model where multiple threads execute within the context of a single process. The threads share the state and resources of a process but are able to execute separately. This enables concurrent execution on a multiprocessor machine.
- Multiple processes (parallelization): In multiprocessing, the child processes are spawned by the parent processes. Initially all the memory processes are shared with the parent, but are duplicated when the data is changed in either the parent or the child processes. Though the communication overhead is considerably higher in comparison to multithreading functionality, most of the operations are safe by default in multiprocessing.

You enable multicore functionality in Design Compiler with the `set_host_options` command. For example, to enable the tool to use six cores to run your processes, execute the following command:

```
set_host_options -max_cores 6
```

The `set_host_options` command supports all `compile_ultra` command options.

Multicore technology supports the following design flows in Design Compiler:

- encapsulated scripts
- retiming
- clock gating
- leakage
- congestion
- incremental compile

Note:
There might be a slowdown in the performance when the number of available cores is less than the number you specify with `set_host_options -max_cores` because predicting machine load is difficult.

If you are in multicore mode, the log file contains an information message similar to the following message:

```
Information: Running optimization using a maximum of 8 cores. (OPT-1500)
```

When running multiple cores, the tool divides the design into many pieces and then runs each piece in a separate process on a separate core. When all processes have finished, the tool reassembles the design and completes the process.

For additional information about multicore technology, see SolvNet article 026392 and the `set_host_options` man page.

Support for Multicorner-Multimode Designs

Designs are often required to operate under multiple modes, such as test or standby mode, and under multiple operating conditions, sometimes referred to as corners. Such designs are known as multicorner-multimode designs. Design Compiler Graphical can analyze and optimize across multiple modes and corners concurrently. The multicorner-multimode feature in Design Compiler Graphical provides compatibility between flows in Design Compiler and IC Compiler.

To define your modes and corners, you use the `create_scenario` command. A scenario definition usually includes commands that specify the TLUPlus libraries, operating conditions, and constraints. For details on setting up multicorner-multimode analysis, see "Optimizing Multicorner-Multimode Designs in Design Compiler Graphical" on page 10-102.

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

To view the licenses that are currently checked out, use the `list_license` command. To determine which licenses are already checked out, use the `license_users` command. For example,

```
dc_shell> license_users
```

```
bill@eng1 Design-Compiler
matt@eng2 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
```

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

Note:
You must have Design Vision license to start the GUI from a dc_shell session.

Enabling License Queuing

Design Compiler has a license queuing functionality that allows your application to wait for licenses to become available if all licenses are in use. To enable this functionality, set the `SNPSLMD_QUEUE` environment variable to true. The following message is displayed:

```
Information: License queuing is enabled. (DCSH-18)
```

When you have enabled the license queuing functionality, you might run into a situation where two processes are waiting indefinitely for a license that the other process owns. Consider the following scenario:

- Two active processes, P1 and P2, are running Design Compiler
- You have the following licenses: Two Design Compiler licenses, a Power Compiler license, and an HDL Compiler license.

For P1 and P2 to be active, they should have checked out one Design Compiler license each. Assume that P1 has acquired the HDL Compiler license and P2 has acquired the Power Compiler license. If P1 and P2 now both attempt to acquire the license held by the other process, both processes wait indefinitely. The `SNPS_MAX_WAITTIME` environment variable and the `SNPS_MAX_QUEUETIME` environment variable help prevent such situations. You can use these variables only if the `SNPSLMD_QUEUE` environment variable is set to true.

The SNPS_MAX_WAITTIME variable specifies the maximum wait time for the first key license that you require, for example, starting dc_shell. Consider the following scenario:

You have two Design-Compiler licenses both of which are in use and are attempting to start a third dc_shell process. The queuing functionality places this last job in the queue for the specified wait time. The default wait time is 259,200 seconds (or 72 hours). If the license is still not available after the predefined time, you might see a message similar to the following:

```
Information: Timeout while waiting for feature ‘Design
Compiler’. (DCSH-17)
```

The `SNPS_MAX_QUEUETIME` variable specifies the maximum wait time for checking out subsequent licenses within the same dc_shell process. You use this variable after you have successfully checked out the first license to start dc_shell. Consider the following scenario:

You have already started Design Compiler and are running a command that requires a DC-Ultra-Features license. The queuing functionality attempts to check out the license within the specified wait time. The default is 28,800 seconds (or eight hours). If the license is still not available after the predefined time, you might see a message similar to the following:

```
Information: Timeout while waiting for feature
'DC-Ultra-Features'. (DCSH-17)
```

As you take your design through the synthesis flow, the queuing functionality might display other status messages as follows:

```
Information: Successfully checked out feature
'DC-Ultra-Opt'. (DCSH-14)
Information: Started queuing for feature
'DC-Ultra-Features'. (DCSH-15)
Information: Still waiting for feature 'DC-Ultra-Features'.
(DCSH-16)
```

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

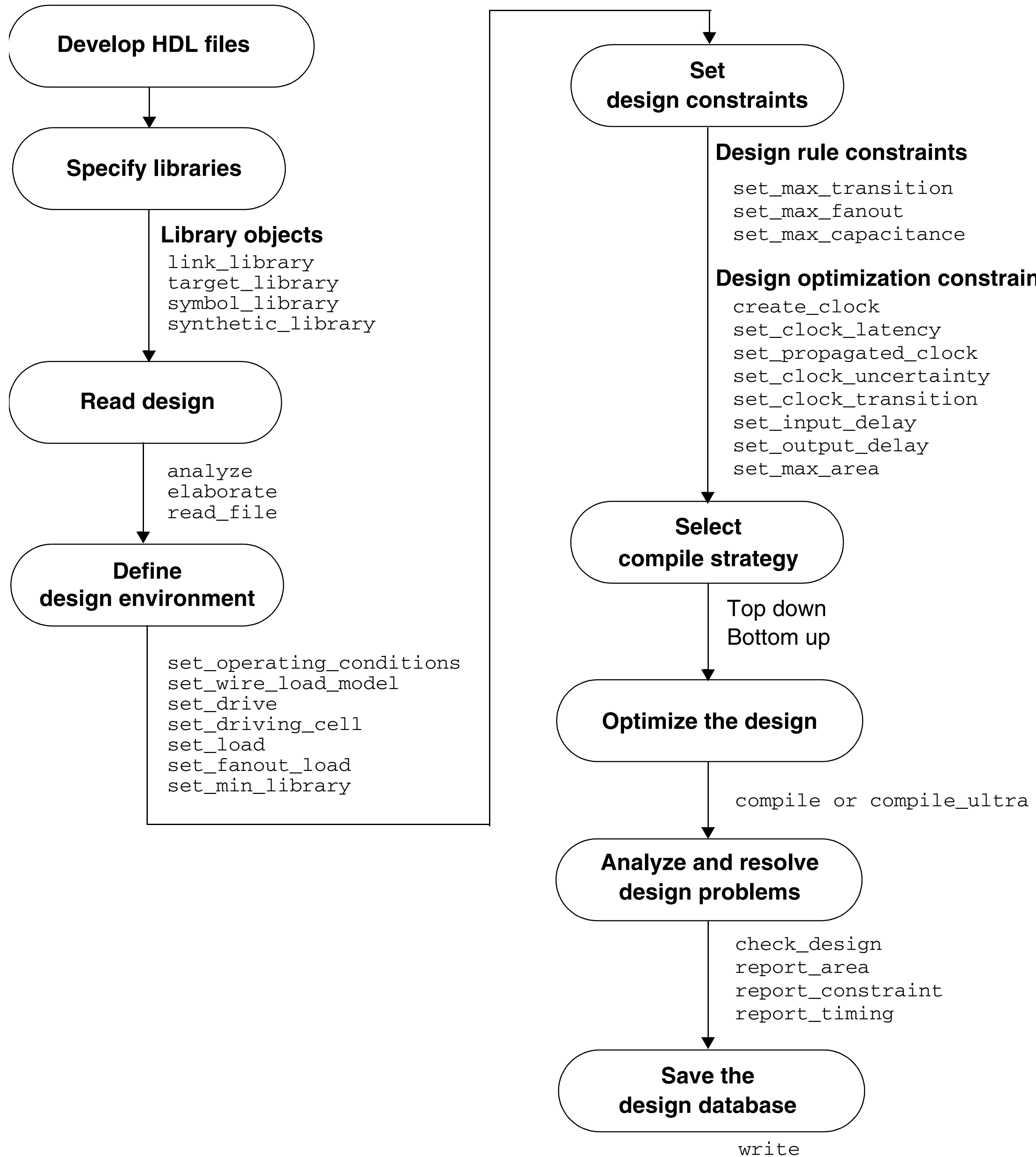
Figure 2-2 shows the basic synthesis flow. You can use this synthesis flow in both the design exploration and design implementation stages of the high-level design flow discussed previously.

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_ultra` command or `compile` command to invoke the Design Compiler synthesis and optimization processes. Several compile options are available with both commands. In particular, the `map_effort` option of the `compile` command can be set to medium or high.

In a default compile, when you are performing design exploration, you use the medium `map_effort` option of the `compile` command. 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.

For designs that have significantly tight timing constraints, you can invoke a single DC Ultra command, `compile_ultra`, for better quality of results (QoR). The command is a push-button solution for timing-critical, high performance designs and encapsulates DC Ultra strategies into a single command.

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 12, “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 2-8 and see the chapter on using script files in the *Design Compiler Command-Line Interface Guide*.

A Design Compiler Session Example

Example 2-2 on page 2-21 shows a simple dctcl script that performs a top-down compile run. 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

```
/* 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 ddc -hierarchy -output Adder16.ddc
```

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 `source` command.

  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 the following command:

  dc_shell> **source run.tcl**

- 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 the following commands:

  % **dc_shell -f run.tcl**

3

Preparing Design Files for Synthesis

Designs (that is, design descriptions) are stored in design files. 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
Synthesis scripts	.con	Constraints
	.scr	Script
Reports and logs	.rpt	Report
	.log	Log
Design database	.ddc	Synopsys internal 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 8-4.

Figure 3-1 Top-Down Compile Directory Structure

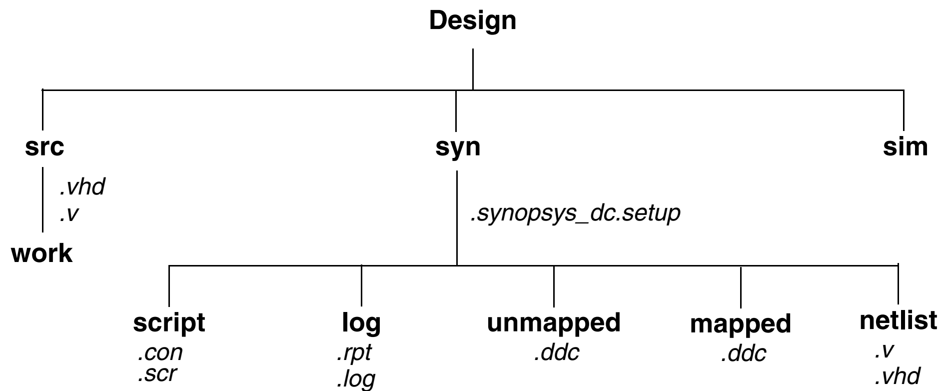
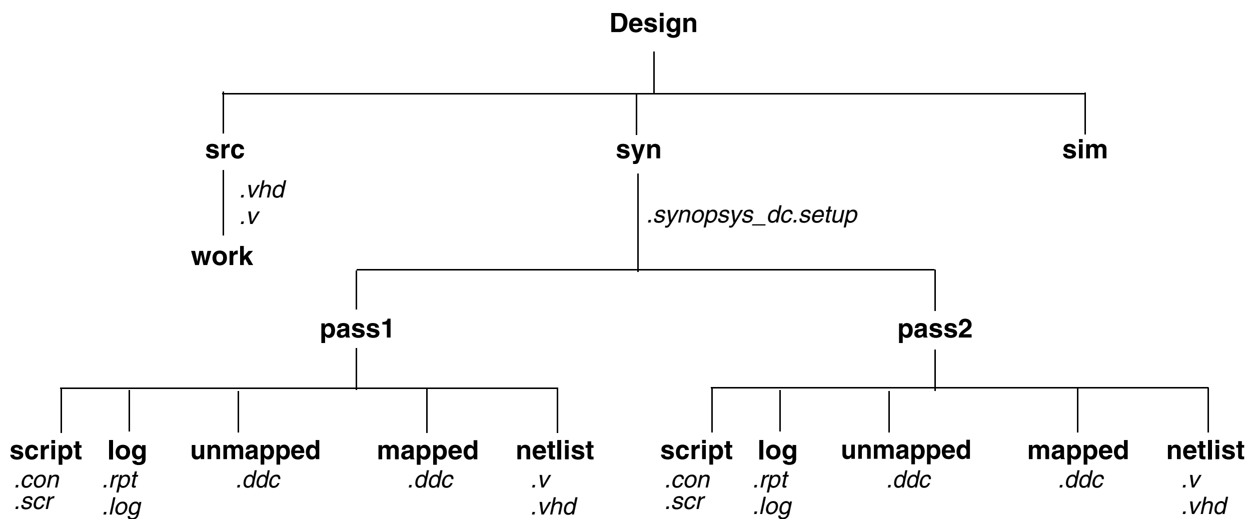


Figure 3-2 Bottom-Up Compile Directory Structure



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

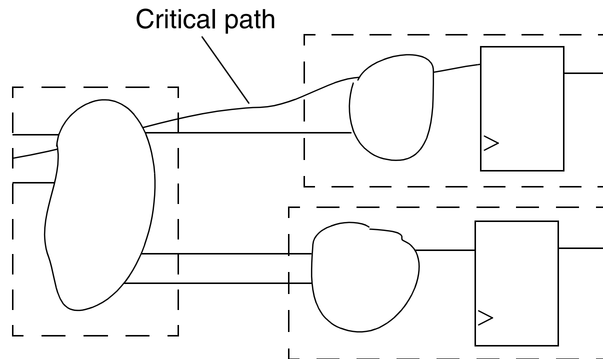
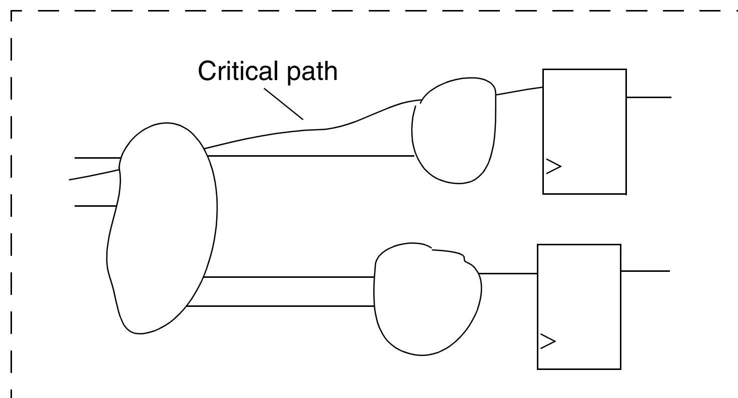


Figure 3-4 shows the same design with no artificial boundaries. In this design, Design Compiler has the flexibility to combine related functions in the combinational clouds.

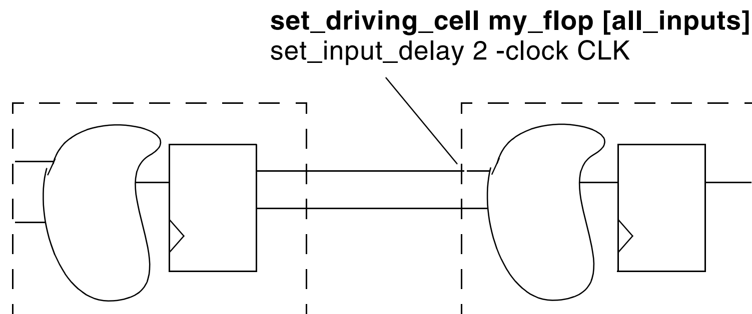
Figure 3-4 Keeping Related Logic in the Same Block



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.

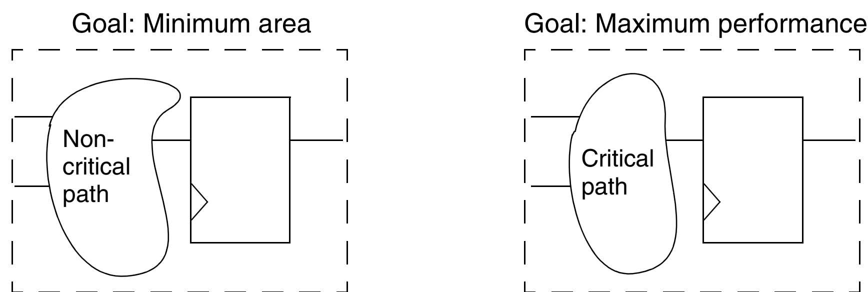
Partitioning by Design Goal

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.

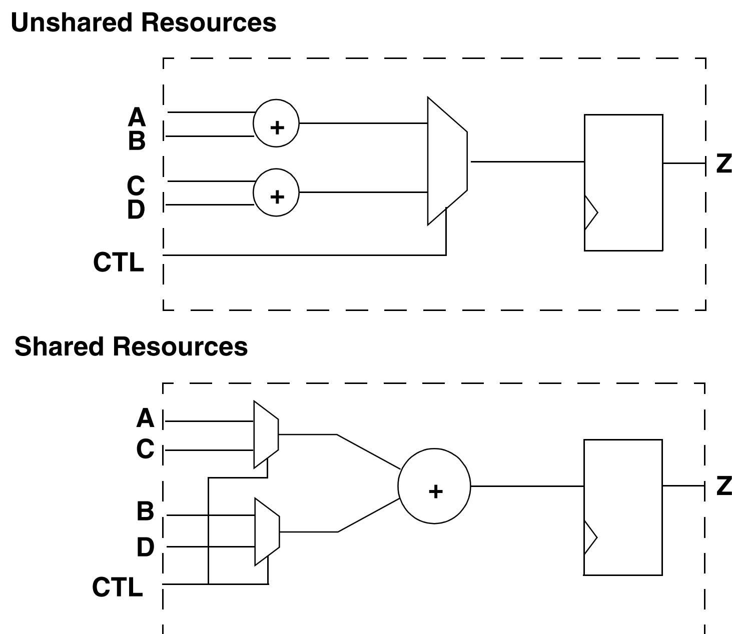
For more information on structuring and flattening, see "Logic-Level Optimization" on page 8-3.

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 3-7 shows possible implementations of a logic example.

Figure 3-7 Keeping Sharable Resources in the Same Process



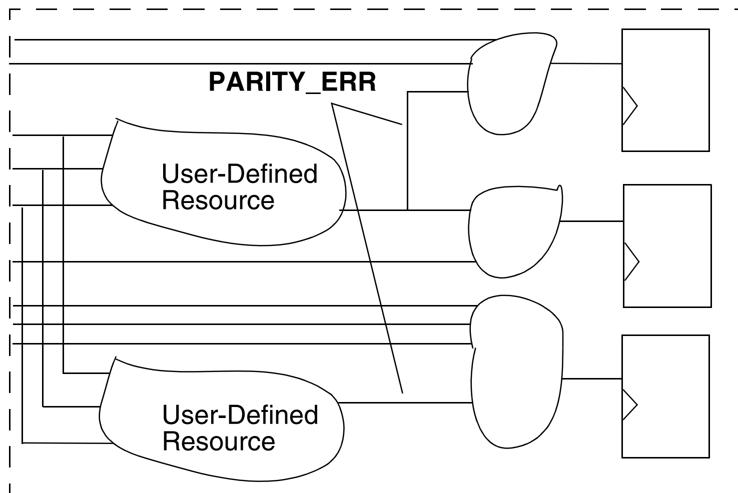
For more information about resource sharing, see the HDL Compiler documentation.

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 3-8 illustrates splitting the load by multiple instantiation when the load on the signal PARITY_ERR is too heavy to meet constraints.

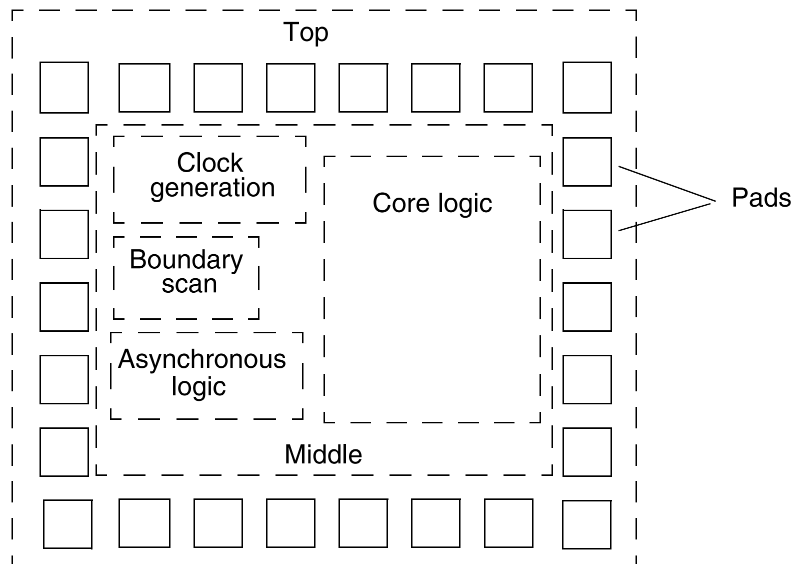
Figure 3-8 Duplicating User-Defined Resources



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 3-9 shows the recommended partitioning for the top level of the design.

Figure 3-9 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 Optimization Reference Manual*.

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 Optimization Reference Manual*.

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

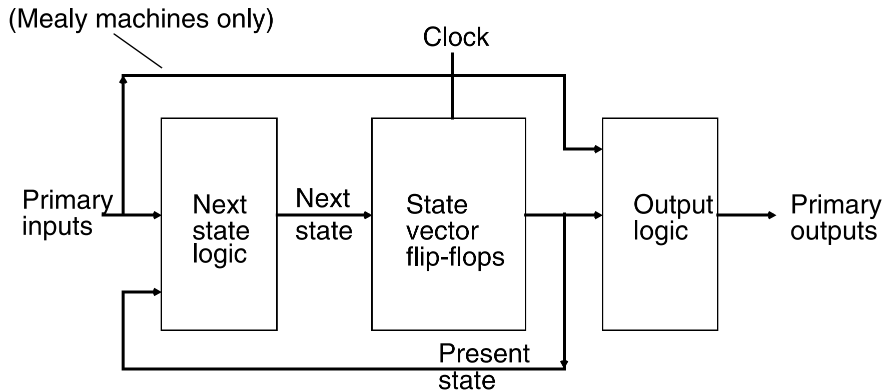
You can specify a state machine by using several different formats:

- 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 casex, casez, and parallel_case information. Design Compiler does not optimize invalid input combinations and mutually exclusive inputs.

Figure 3-10 shows the architecture for a finite state machine.

Figure 3-10 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 Optimization Reference Manual*.

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 correct simulation results,

- Use delayed (nonblocking) assignments within sequential Verilog always blocks or VHDL processes
- Use immediate (blocking) 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 my_design
entity my_design is
   port (a,b : in std_logic_vector(WIDTH-1 downto 0);
         c: out std_logic_vector(WIDTH-1 downto 0));
end my_design;

-- Define and use local constant in entity my_design
entity my_design 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 my_design;
```

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); I 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
- Library-Aware Mapping and Synthesis
- 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 supports logic libraries that use nonlinear delay models (NLDMs), Composite Current Source (CCS) models (either compact or noncompact), or both. Design Compiler automatically selects the timing model to use based on the contents of the logic libraries. If the logic libraries contain only CCS models, the tool uses CCS models. If the libraries contain nonlinear delay models (NLDMs) or both NLDM and Composite Current Source (CCS) models, the tool uses the NLDM models. For optimal runtime and the level of accuracy, the tool may not use all of the CCS data during logic synthesis and pre-route optimizations.

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 5-40.

- Resolving cell references

  The technology libraries that Design Compiler uses to resolve cell references are called link libraries.

  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 5-13.

- 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	target_library	{“your_library.db”}	.db
Link library	link_library	{“*”, “your_library.db”}	.db
Symbol library	symbol_library	{“your_library.sdb”}	.sdb
DesignWare library	synthetic_library	{}	.sldb

Specifying Technology Libraries

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

Target 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.

Use the `target_library` variable to specify the target library.

Link Library

Design Compiler uses the link library to resolve references. For a design to be complete, it must connect to all the library components and designs it references. This process is called linking the design or resolving references.

During the linking process, Design Compiler uses the `link_library` system variable, the `local_link_library` attribute, and the `search_path` system variable to resolve references. These variables and attribute are described below:

- `link_library` variable

  The `link_library` variable specifies a list of libraries and design files that Design Compiler can use to resolve references. When you load a design into memory, Design Compiler also loads all libraries specified in the `link_library` variable.

Because the tool loads the libraries while loading the design, rather than during the link process, the memory usage and runtime required for loading the design might increase. However, the advantage is that you know immediately whether your design can be processed with the available memory.

An asterisk in the value of the `link_library` variable specifies that Design Compiler should search memory for the reference.

- `local_link_library` attribute

  The `local_link_library` attribute is a list of design files and libraries added to the beginning of the `link_library` variable during the linking process. Design Compiler searches files in the `local_link_library` attribute first when it resolves references.

- `search_path` variable

  If Design Compiler does not find the reference in the link libraries, it searches in the directories specified by the `search_path` variable, described in "Specifying a Library Search Path" on page 4-8. For more information on resolving references, see "Linking Designs" on page 5-13.

  If you do technology translation, add the standard cell library for the existing mapped gates to the `link_library` and the standard cell library being translated to the `target_library`.

Your `target_library` specification should only contain those standard cell libraries that you want Design Compiler to use when mapping your design's standard cells. Standard cells are cells such as combinational logic and registers. Your `target_library` specification should not include any DesignWare libraries or macro libraries such as pads or memories. The `target_library` is a subset of the `link_library` and listed first in your list of link libraries, as shown in Example 4-1. This example includes the additional_link_lib_files user created variable to simplify the link library definition.

Example 4-1 Setting the Target, Synthetic, and Link Libraries

```
set target_library [list of standard cell library files for mapping]
set synthetic_library [list of sldb files for designware, and so on]
set additional_link_lib_files [list of additional libraries for linking: pads, macros,
                               and so on]
set link_library [list * $target_library $additional_link_lib_files \
                         $synthetic_library]
```

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.

Design Compiler uses the first technology library found in the `link_library` variable as the main library. It 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

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

If you are performing simultaneous minimum and maximum timing analysis, the logic libraries specified by the `link_library` variable are used for both maximum and minimum timing information, unless you specify separate minimum timing libraries by using the `set_min_library` command. The `set_min_library` command associates minimum timing libraries with the maximum timing libraries specified in the `link_library` variable. For example,

```
dc_shell> set link_library "* maxlib.db"
dc_shell> set_min_library maxlib.db -min_version minlib.db
```

To find out which libraries have been set to be the minimum and maximum libraries, use the `list_libs` command. In the generated report, the letter "m" appears next to the minimum library and the letter "M" appears next to the maximum library.

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 documentation.

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, where $SYNOPSYS is the path to the installation directory. 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
```

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
Logical 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
1
```

To set the `disable_timing` attribute on the Z pin of the AND4 cell in the my_lib library, enter the following command:

```
dc_shell> set_disable_timing [get_pins my_lib/AND4/Z]
1
```

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
1
```

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

```
dc_shell> remove_attribute MY_LIB/INV_HD dont_use
MY_LIB/INV_HD
```

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
1
```

Use the `remove_attribute` command to remove cell preferences.

```
dc_shell> remove_attribute MY_LIB/INV_LD preferred
MY_LIB/INV_LD
```

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
1
```

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

```
dc_shell> remove_attribute class/IV hold_preferred
class/IV
```

Library-Aware Mapping and Synthesis

You can characterize (or analyze) your target technology library and create a pseudolibrary called ALIB, which has mappings from Boolean functional circuits to actual gates from the target library. Design Compiler reads the ALIB file during compile. The ALIB file provides Design Compiler with greater flexibility and a larger solution space to explore tradeoffs between area and delay during optimization. Use the `compile_ultra` command or DC Ultra optimization to get the maximum benefit from the ALIB library.

Library characterization occurs during the initial stage of compile. Because it can take 5-10 minutes for Design Compiler to characterize each technology library, it is recommended that you generate the ALIB file when you install Design Compiler, and store it in a single repository so that multiple users can share the library.

Generating the ALIB file

Use the `alib_analyze_libs` command to generate the ALIB library corresponding to your target technology library. The tool creates a release-specific subdirectory in the location specified by the `alib_library_analysis_path` variable and stores the generated ALIB files in this directory. For example, the following sequence of commands creates the x.db.alib file for the target library x.db and stores it in /remote/libraries/alib/alib-51.

```
dc_shell> set target_library "x.db"
dc_shell> set link_library "x.db"
dc_shell> set alib_library_analysis_path "/remote/libraries/alib"
dc_shell> alib_analyze_libs
```

Design Compiler creates the alib-51 directory for version control; each release has a different version.

Using the ALIB library

To load the previously generated ALIB library, use the `alib_library_analysis_path` variable to point to the location of the file. For example,

```
dc_shell> set alib_library_analysis_path "remote/libraries/alib"
```

During compile, if no pre-generated ALIB libraries exist, Design Compiler performs library characterization automatically. It generates the ALIB library in the location specified by the `alib_library_analysis_path` variable. If you have not set this variable, the ALIB library is stored in the current working directory. The tool uses this ALIB library for subsequent runs.

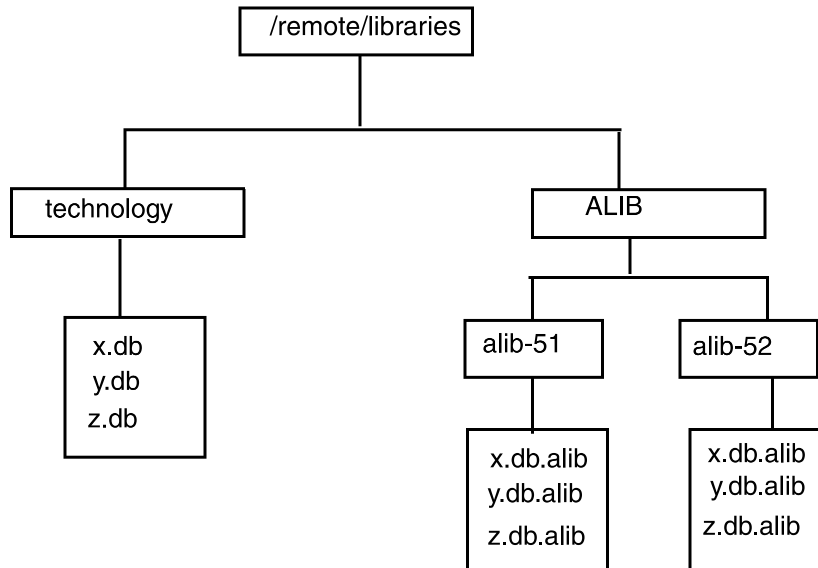
It is recommended that you generate the ALIB library for your target technology library when you install Design Compiler. Create a directory structure similar to the one you use for storing technology libraries as shown in Figure 4-1. If you re-install Design Compiler, you must regenerate the ALIB library. A new subdirectory is created within the /remote/libraries/alibs directory.

Note:

In version Y-2006.06-SP4 and later, Design Compiler reads ALIB files in the alib-52 directory. Design Compiler automatically generates the new ALIB files when you run the Y-2006.06-SP4 version for the first time.

If you are using the pre-built ALIB file in a central location, create new ALIB files before compiling.

Figure 4-1 Directory Structure for ALIB library



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_libs` command to see the path for each library in memory.

Saving Libraries

The `write_lib` command saves (writes to disk) a compiled library in the Synopsys database or VHDL format. Use the `write_link_lib` command to write out the shell commands to save the current link library settings for design instances.

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 or Verilog HDL.

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.

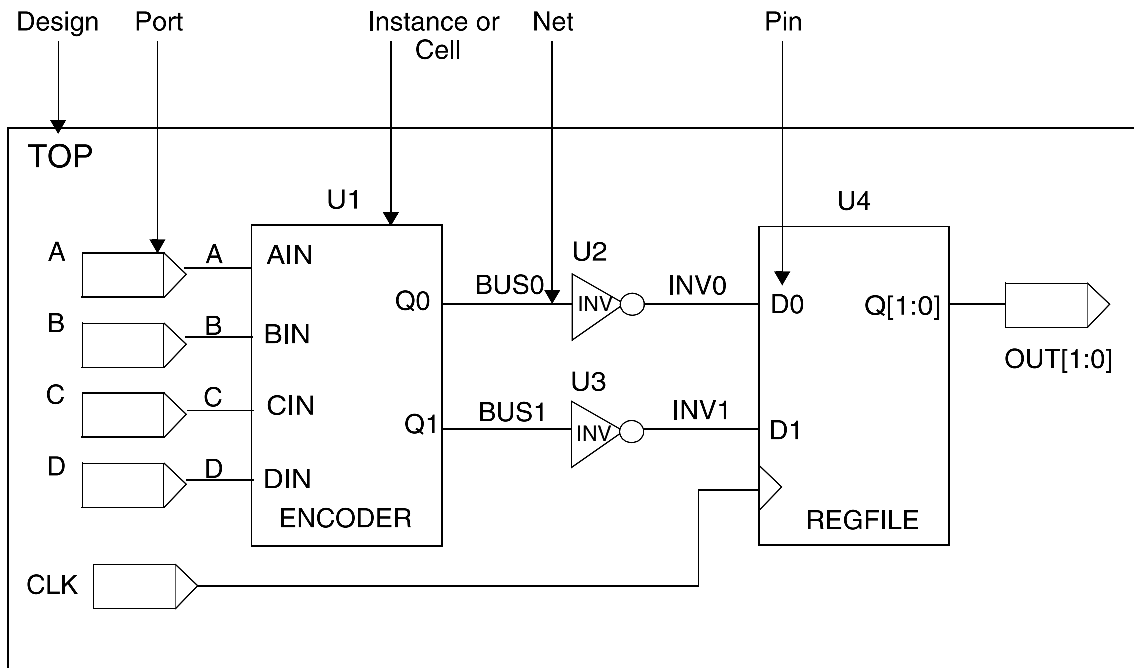
Hierarchical Designs

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.

Design Objects

Figure 5-1 shows the design objects in a design called TOP. Synopsys commands, attributes, and constraints are directed toward specific design objects.

Figure 5-1 Design Objects



Design: {TOP, ENCODER, REGFILE}

Reference: {ENCODER, REGFILE, INV}

Instance: {U1, U2, U3, U4}

Design

A design consists of instances, nets, ports, and pins. It can contain subdesigns and library cells. In Figure 5-1, the designs are TOP, ENCODER, and REGFILE. 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.

Reference

A reference is a library component or design that can be used as an element in building a larger circuit. The structure of the reference can be a simple logic gate or a more complex design (a RAM core or CPU). A design can contain multiple occurrences of a reference; each occurrence is an instance.

References enable you to optimize every cell (such as a NAND gate) in a single design without affecting cells in other designs. The references in one design are independent of the same references in a different design. In Figure 5-1, the references are INV, ENCODER, and REGFILE.

Instance or Cell

An instance is 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.

A unique instance of a design within another design is called a hierarchical instance. A unique instance of a library cell within a design is called a leaf 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. In Figure 5-1, the instances are U1, U2, U3, and U4.

Ports

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

Pins

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

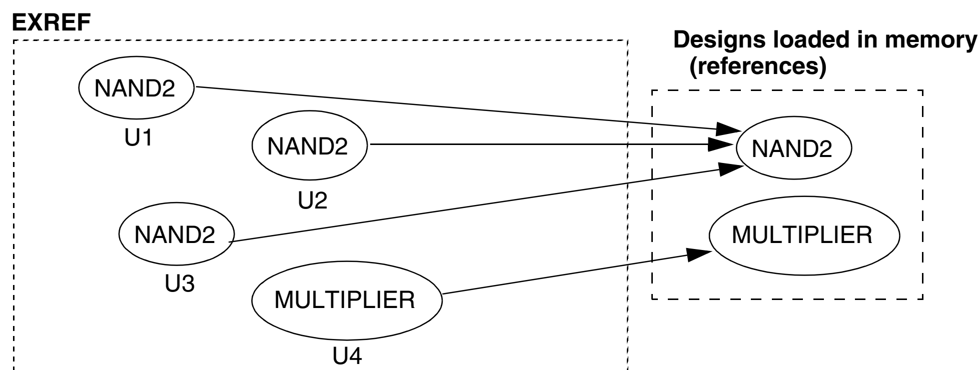
Nets

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

Relationship Between Designs, Instances, and References

Figure 5-2 shows the relationships among designs, instances, and references.

Figure 5-2 Instances and 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 5-13.

Reporting References

You can use the `report_reference` command to report information about all references in the current instance or current design. Use the `-hierarchy` option to display information across the hierarchy in the current instance or current design.

Using Reference Objects

When you use the `get_references` command, Design Compiler returns a collection of instances that have the specified reference, and you operate on the instances.

For example, the following command returns a collection of instances in the current design that have the reference AN2:

```
dc_shell> get_references AN2
{U2 U3 U4}
```

To see the reference names, use the following command:

```
dc_shell> report_cell [get_references AN*]


Cell                       Reference       Library             Area
Attributes
--------------------------------------------------------------------------
U2                          AN2             lsi_10k          2.000000
U3                         AN2             lsi_10k          2.000000
U4                         AN2             lsi_10k          2.000000
U8                         AN3             lsi_10k          2.000000
```

Reading Designs

Design Compiler can read designs in the formats listed in Table 5-1.

Table 5-1 Supported Input Formats

Format	Description
.ddc	Synopsys internal database format (recommended)
.db	Synopsys internal database format
equation	Synopsys equation format
pla	Berkeley (Espresso) PLA format
st	Synopsys State Table format
Verilog	IEEE standard Verilog (see the HDL Compiler documentation)
VHDL	IEEE standard VHDL (see the HDL Compiler documentation)

Commands for Reading Design Files

Design Compiler provides the following ways to read design files:

- The `analyze` and `elaborate` commands
- The `read_file` command

Using the analyze and elaborate Commands

The `analyze` command does the following:

- 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. After a design is analyzed, you must reanalyze it only when you change it.

Use options to the `analyze` command as follows:

Table 5-2 Using the analyze Command Options

To do this	Use this
Store design elements in a library other than the work library	`-library` By default, the `analyze` command stores all output in the work library.
Specify the format of the files to be analyzed	`-vhdl` or `-verilog`
Specify a list of files to be analyzed	`file_list`

The `elaborate` command does the following:

- Translates the design into a technology-independent design (GTECH) from the intermediate files produced during analysis
- Allows changing of parameter values defined in the source code
- Allows VHDL architecture selection
- Replaces the HDL arithmetic operators in the code with DesignWare components
- Automatically executes the `link` command, which resolves design references

Use options to the `elaborate` command as follows:

Table 5-3 Using the elaborate Command Options

To do this	Use this
Specify the name of the design to be built (the design can be a Verilog module, a VHDL entity, or a VHDL configuration)	`-design_name`
Find the design in a library other than the work library (the default)	`-library`
Specify the name of the architecture	`-architecture`
Automatically reanalyze out-of-date intermediate files if the source can be found	`-update`
Specify a list of design parameters	`-parameters`

For more information about the analyze and elaborate commands, see the man pages and *HDL Compiler for Verilog User Guide* or the *HDL Compiler for VHDL User Guide*.

Using the read_file Command

The read_file command does the following:

- Reads several different formats
- Performs the same operations as analyze and elaborate in a single step
- Creates .mr and .st intermediate files for VHDL
- Does not execute the link command automatically (see “Linking Designs” on page 5-13)
- Does not create any intermediate files for Verilog (However, you can have the read_file command create intermediate files by setting the hdlin_auto_save_templates variable to true)

For designs in memory, Design Compiler uses the naming convention *path_name* / *design*.ddc. 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.

If you do not specify the design format, the read_file command will infer the format based on the file extension. If no known extension is used, the tool assumes the .ddc format. Supported extensions for automatic inference are not case sensitive. The read_file command is able to read compressed files for all formats except the .ddc format, which is compressed internally as it is written. To enable the tool to automatically infer the file format for compressed files, use the following naming structure: <filename>.<format>.gz.

The following formats are supported:

- ddc format: Uses the .ddc extension
- db format: Uses the .db, .sldb, .sdb, .db.gz, .sldb.gz, and .sdb.gz extensions
- Verilog format: Uses the .v, .verilog, .v.gz, and .verilog.gz extensions
- SystemVerilog: Uses the .sv, .sverilog, .sv.gz, and .sverilog.gz extensions
- VHDL: Uses the .vhd, .vhdl, vhd.gz, and .vhdl.gz extensions

Use options to the `read_file` command as follows:

Table 5-4 Using the read_file Command Options

To do this	Use this
Specify a list of files to be read	`file_list`
Specify the format in which a design is read You can specify any input format listed in Table 5-1.	`-format`
Store design elements in a library other than the work library (the default) when reading VHDL design descriptions	`-library`
Read a Milkyway ILM view	`-ilm`
Specify that the design being read is a structural or gate-level design when reading Verilog or VHDL designs	`-netlist -format\ verilog|vhdl`[1]
Specify that the design being read is an RTL design when reading Verilog or VHDL designs	`-rtl -format\ verilog|vhdl`[2]

1. *The -netlist option is optional when you read a Verilog design.*
2. *The -rtl option is optional when you read a Verilog design.*

Table 5-5 summarizes the differences between using the `read_file` command and using the `analyze` and `elaborate` commands to read design files.

Table 5-5 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 `elaborate` command line. Thus for parameterized designs, you can use the `analyze` and `elaborate` commands to build a new design with nondefault values.
Architecture	Cannot specify the architecture to be elaborated	Allows you to specify architecture to be elaborated.
Linking designs	Must use the `link` command to resolve references	The `elaborate` command executes the `link` command automatically to resolve references.

Reading HDL Designs

Use one of the following methods to read HDL design files:

- The `analyze` and `elaborate` commands

  To use this method, analyze the top-level design and all subdesigns in bottom-up order and then elaborate the top-level design and any subdesigns that require parameters to be assigned or overwritten.

  For example, enter

  ```
  dc_shell> analyze -format vhdl -lib -work RISCTYPES.vhd
  dc_shell> analyze -format vhdl -lib -work {ALU.vhd STACK_TOP.vhd
                                        STACK_MEM.vhd...}
  dc_shell> elaborate RISC_CORE -arch STRUCT -lib WORK -update
  ```

- The `read_file` command

  For example, enter

  ```
  dc_shell> read_file -format verilog RISC_CORE.v
  ```

- The `read_verilog` or `read_vhdl` command

For example, enter

```
dc_shell> read_verilog RISC_CORE.v
```

You can also use the `read_file -format VHDL` and `read_file -format verilog` commands.

Reading .ddc Files

To read the design data from a .ddc file, use the `read_ddc` command or the `read_file -format ddc` command. For example,

```
dc_shell> read_ddc design_file.ddc
```

Note:
The .ddc format is backward compatible (you can read a .ddc file that was generated with an earlier software version) but not forward compatible (you cannot read a .ddc file that was generated with a later software version).

Reading .db Files

Although you can use the .db format, it is recommended that you use the .ddc format. To read in a .db file, use the `read_db` command or the `read_file -format db` command.

For example,

```
dc_shell> read_db design_file.db
```

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.

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.ddc
A (*)

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

The asterisk (*) next to design A shows that A is the current design. File B.ddc contains both designs B and C.

Setting the Current Design

You can set the current design (the design you are working on) 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 ddc MY_DESIGN.ddc
  Reading ddc file '/designs/ex/MY_DESIGN.ddc'
  Current design is '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 the following command:

```
dc_shell> printvar current_design
current_design = "test"
```

Using the current_design Command

You should avoid writing scripts that use a large number of `current_design` commands, such as in a loop. Using a large number of `current_design` commands can increase runtime. For more information, see the *Design Compiler Command-Line Interface Guide*, chapter 5.

Several commands accept instance objects—that is, cells at a lower level of hierarchy. You can operate on hierarchical designs from any level in the design without using the `current_design` command. The enhanced commands are listed below:

- Netlist editing commands.

  For more information, see “Editing Designs” on page 5-31.

- The `ungroup`, `group`, and `uniquify` commands

  For more information, see “Removing Levels of Hierarchy” on page 5-25 and “Uniquify Method” on page 8-13.

- The `set_size_only` command

  The command sets a list of attributes on specified leaf cells so sizing optimizations can be performed on these cells during compile.

- The `change_link` command

  For more information, see “Changing Design References” on page 5-15.

Linking Designs

For a design to be complete, it must connect to all the library components and designs it references. This process is called linking the design or resolving references.

Design Compiler uses the `link` command to resolve references. The `link` command uses the `link_library` and `search_path` system variables and the `local_link_library` attribute to resolve design 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.
   a. Design Compiler first searches the libraries and design files defined in the current design’s `local_link_library` attribute

b. If an asterisk is specified in the value of the `link_library` variable, Design Compiler searches in memory for the reference.

c. Design Compiler then searches the libraries and design files defined in the `link_library` variable.

3. If it does not find the reference in the link libraries, it searches in the directories specified by the `search_path` variable. See “Locating Designs by Using a Search Path” on page 5-15.

4. It links (connects) the located references to the design.

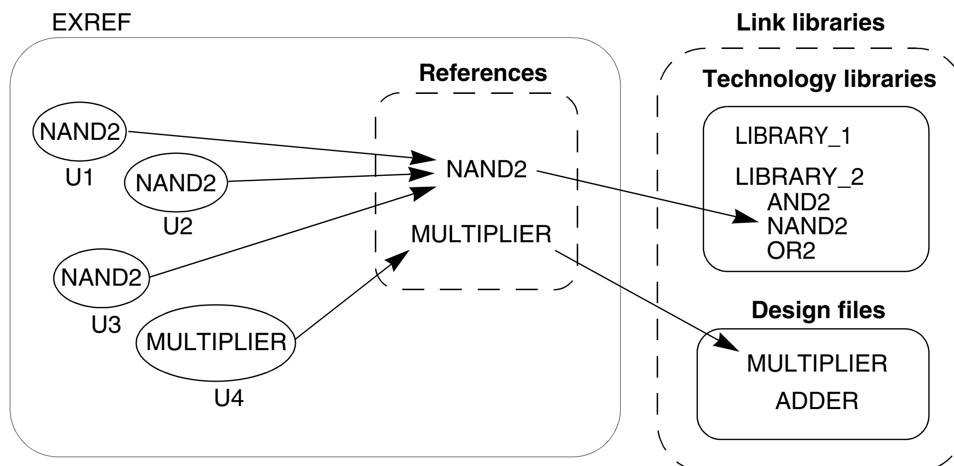
Note:
In a hierarchical design, Design Compiler considers only the top-level design’s local link library. It ignores local link libraries associated with the subdesigns.

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 5-3 show the connections that the linking process added between the instances, 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 5-3 Resolving References



Locating Designs by 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. 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. By default, the search path includes the current working directory and $SYNOPSYS/libraries/syn, where $SYNOPSYS is the path to the installation directory. 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.ddc
{/usr/designers/example/my_design.ddc}
```

To specify other directories in addition to the default search path, use one following command:

```
dc_shell> lappend search_path project
```

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 running the `change_link` command, you must run the design with the `link` command.

The `change_link` command accepts instance objects, that is, cells at a lower level in the hierarchy. Additionally, you can use the `-all_instances` option when any cell in the `object_list` is an instance at a lower level in the hierarchy and its parent cell is not unique. All similar cells in the same parent design are automatically linked to the new reference design. You do not have to change the current design to change the link for such cells. If none of the cells in `object_list` is an instance at a lower level of the hierarchy or the parent cell is unique, you do not have to use the `-all_instances` option.

Example 1

The following command shows how cells U1 and U2 are linked from the current design to MY_ADDER:

```
dc_shell> copy_design ADDER MY_ADDER
dc_shell> change_link {U1 U2} MY_ADDER
```

Example 2

The following command changes the link for cell U1, which is at a lower level in the hierarchy:

```
dc_shell> change_link top/sub_inst/U1 lsi_10k/AN3
```

Example 3

This example shows how you can use the `-all_instances` option to change the link for inv1, when its parent design, bot, is instantiated multiple times. The design bot is instantiated twice: mid1/bot1 and mid1/bot2.

```
dc_shell> change_link -all_instances mid1/bot1/inv1 lsi_10k/AN3

Information: Changed link for all instances of cell 'inv1'
in subdesign 'bot'. (UID-193)

dc_shell> get_cells -hierarchical -filter "ref_name == AN3"

{mid1/bot1/inv1 mid1/bot2/inv1}
1
```

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 in Table 5-6 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 collection that can be used as input to another dc_shell command.

Table 5-6 lists the commands and the actions they perform.

Table 5-6 Commands to Access Design Objects

Object	Command	Action
Instance	list_instances report_cell	Lists instances and their references. Displays information about instances.
Reference	report_reference	Displays information about references.
Port	report_port report_bus all_inputs all_outputs	Displays information about ports. Displays information about bused ports. Returns all input ports. Returns all output ports.
Net	report_net report_bus	Displays information about nets. Displays information about bused nets.
Clock	report_clock all_clocks	Displays information about clocks. Returns all clocks.
Register	all_registers	Returns all registers.
Collections	get_*	Returns a collection of cells, designs, libraries and 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
1
```

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
Current instance is the top-level of the design 'Count_16'
```

The `current_instance` variable points to the current instance. To display the current instance, enter the following command:

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

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 the `get_*` command. 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.ddc:Count_16/U1/U5
1
```

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
1
dc_shell> list_designs -show_file

/work_dir/mapped/test.ddc
test (*) test_DW01_inc_16_0 test_DW02_mult_16_16_1

/work_dir/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 5-31.

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 test test_new
Information: Copying design /designs/test.ddc:to designs/
test.ddc:test_new
1
dc_shell> list_designs -show_file

/designs/test.ddc
test (*) test_new
```

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
Information: Copying design /designs/COMP.ddc:COMP to
     designs/COMP.ddc:COMP1
1

dc_shell> change_link U1 COMP1
Performing change_link on cell 'U1'.
1

dc_shell> copy_design COMP COMP2
Information: Copying design /designs/COMP.ddc:COMP to
                designs/COMP.ddc:COMP2
1

dc_shell> change_link U2 COMP2
Performing change_link on cell 'U2'.
1
```

Renaming Designs

Use the `rename_design` command to rename a design in memory. You can assign a new name to a design or move a list of designs to a file. To save a renamed file, use the `write` command. Use options to the `rename_design` command as follows:

Table 5-7 Using the rename_design Command Options

To do this	Use this
Specify a list of designs to be renamed	*-design_list*
Specify the new name of the design	*-target_name*
Specify a string to be prefixed to the design name	`-prefix`
Specify a string to be appended to the design name	`-postfix`
Relink cells to the renamed reference design	`-update_links`

In the following example, the `list_designs` command is used to show the design before and after you use the `rename_design` command:

```
dc_shell> list_designs -show_file

/designs/test.ddc
test(*) test_new
```

```
1

dc_shell> rename_design test_new test_new_1
Information: Renaming design /designs/test.ddc:test_new to
      /designs/test.ddc:test_new_1
1

dc_shell> list_designs -show_file

/designs/test.ddc
test (*) test_new test_new_1
1
```

You can use the `-prefix`, `-postfix,` and `-update_links` options to rename designs and update cell links for the entire design hierarchy. For example, the following script prefixes the string NEW_ to the name of the design D and updates links for its instance cells:

```
dc_shell> get_cells -hier -filter "ref_name == D"

{b_in_a/c_in_b/d1_in_c b_in_a/c_in_b/d2_in_c}

dc_shell> rename_design D -prefix NEW_ -update_links
Information: Renaming design /test_dir/D.ddc:D to
                              /test_dir/D.ddc:NEW_D. (UIMG-45)

dc_shell> get_cells -h -filter "ref_name == D"
# no  such cells!

dc_shell> get_cells -h -filter "ref_name == NEW_D"
{b_in_a/c_in_b/d1_in_c b_in_a/c_in_b/d2_in_c}
```

Cells `b_in_a/c_in_b/d1_in_c` and `b_in_a/c_in_b/d2_in_c` instantiate design D. After you have run the `rename_design D -prefix NEW -update_links` command, the instances are relinked to the renamed reference design NEW_D.

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

You use the `group` command to group 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, use its arguments and options as follows:

Table 5-8 Using the group Command

To do this	Use this
Specify a list of cells to be grouped into the new subdesign. When the parent design is unique, the list can include cells from a lower level in the hierarchy; however, these cells should be at the same level of hierarchy in relation to one another. To exclude cells from the specified list use the `-except` option.	Provide a list of cells as an argument to the `group` command
Specify the name of the new subdesign	`-design_name`

Table 5-8 Using the group Command

To do this	Use this
Specify the new instance name (optional) If you do not specify an instance name, Design Compiler creates one for you. The created instance name has the format U*n,* where *n* is an unused cell number (for example, U107).	`-cell_name`

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

The following examples illustrate how to use the `group` command.

Example 1

To group two cells into a new design named SAMPLE with an instance name U, enter

```
dc_shell> group {u1 u2} -design_name SAMPLE -cell_name U
```

Example 2

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
```

Example 3

In the following example, three cells— bot1, foo1, and j —are grouped into a new subdesign named SAMPLE, with an instance name U1. The cells are at a lower level in the hierarchy and at the same hierarchical level; the parent design is unique.

```
dc_shell> group {mid1/bot1 mid1/foo1 mid1/j}
                    -cell_name U1 -design_name SAMPLE
```

The preceding command is equivalent to issuing the following two commands:

```
dc_shell> current_design mid
dc_shell> group {bot1 foo1 j} -cell_name U1 -design_name SAMPLE
```

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, use options to the `group` command as follows:

Table 5-9 Using the group Command Options

To do this	Use this
Specify one of the following component types:	
Bused gates	`-hdl_bussed`
Combinational logic	`-logic`
Finite state machines	`-fsm`
HDL blocks	`-hdl_all_blocks` `-hdl_block` *`block_name`*
PLA specifications	`-pla`
Specify the name of the new subdesign	`-design_name`
Optionally, specify the new instance name If you do not specify an instance name, Design Compiler creates one for you. The created instance name has the format U*n,* where *n* is an unused cell number (for example, U107).	`-cell_name`

Note:
You cannot use the `-design_name` and `-cell_name` options with the `hdl_all_blocks` or `hdl_bussed` option.

Example 1

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
```

Example 2

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.

Use the following arguments and options to the `ungroup` command:

Table 5-10 Using the ungroup Command Options

To do this	Use this
Specify a list of cells to be ungrouped When the parent design is unique, the list can include cells from a lower level in the hierarchy (that is, the `ungroup` command can accept instance objects)	Provide a list of cells as an argument to the `ungroup` command
Ungroup all cells in the current design or current instance	`-all`
Ungroup each cell recursively until all levels of hierarchy within the current design (instance) are removed	`-flatten`
Ungroup cells recursively starting at any hierarchical level You must specify a number for this option: 1, 2, 3, and so on. A value of 1 indicates that cells from the current design are to be ungrouped. The cells that are at the level specified by the `-start_level` option are included in the ungrouping. Additionally, when you use this option, the current instance cannot be set.	`-start_level` *number*

Table 5-10 Using the ungroup Command Options

To do this	Use this
Specify the prefix to use in naming ungrouped cells If you do not specify a prefix, Design Compiler uses the prefix *cell_to_be_ungrouped/ old_cell_name {number}*.	`-prefix` *`prefix_name`*
Ungroup subdesigns with fewer leaf cells than a specified number	`-small` *`number`*

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

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

Example 1

To ungroup a list of cells, enter

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

Example 2

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

```
dc_shell> ungroup U1 -prefix "U1:”
```

Example 3

To completely collapse the hierarchy of the current design, enter

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

Example 4

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
```

Example 5

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}
```

Example 6

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

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

Example 7

This example illustrates how the `ungroup` command can accept instance objects (cells at a lower level of hierarchy) when the parent design is unique. In the example, MID1/BOT1 is a unique instantiation of design BOT. The command ungroups the cells MID1/BOT1/FOO1 and MID1/BOT1/FOO2.

```
dc_shell> ungroup {MID1/BOT1/FOO1 MID1/BOT1/FOO2}
```

The preceding command is equivalent to issuing the following two commands:

```
dc_shell> current_instance MID1/BOT1
dc_shell> ungroup {FOO1 FOO2}
```

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

To automatically ungroup hierarchies during optimization, you can use the `compile_ultra` command or the `-auto_ungroup` option of the `compile` command. Design Compiler provides two options to automatically ungroup hierarchies: area-based auto-ungrouping and delay-based auto-ungrouping.

By default, the `compile_ultra` command performs delay-based auto-ungrouping. It ungroups hierarchies along the critical path and is used essentially for timing optimization. In addition, the command performs area-based auto-ungrouping before initial mapping. The tool estimates the area for unmapped hierarchies and removes small subdesigns; the goal is to improve area and timing quality of results.

To use the auto-ungrouping capability of the `compile` command, enter

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

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

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. For more information on automatic ungrouping, see the *Design Compiler Optimization Reference Manual.*

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 5-11 Preserving Hierarchical Pin Timing Constraints

Compile flow	Effect on hierarchical pin timing constraints
Ungrouping a hierarchy before optimization by using ungroup	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 ungroup_preserve_constraints 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 compile -ungroup_all or set_ungroup followed by compile	Timing constraints placed on hierarchical pins are not preserved. To preserve timing constraints, set the auto_ungroup_preserve_constraints variable to true.
Automatically ungrouping a hierarchy during optimization, that is, by using the compile_ultra command or compile -auto_ungroup area|delay	Design Compiler does not ungroup the hierarchy. To make Design Compiler ungroup the hierarchy and preserve timing constraints, set the auto_ungroup_preserve_constraints variable to true.

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 5-12 Design Editing Tasks and Commands

Object	Task	Command
Cells	Create a cell Delete a cell	`create_cell` `remove_cell`

Nets	Create a net
	Connect a net
	Disconnect a net
	Delete a net
		`create_net`
		`connect_net`
		`disconnect_net`
		`remove_net`

Ports	Create a port
	Delete a port
		`create_port`
		`remove_port`
		`remove_unconnected_port`

Pins	Connect pins	`connect_pin`
Buses	Create a bus Delete a bus	`create_bus` `remove_bus`

For unique designs, these netlist editing commands accept instance objects—that is, cells at a lower level of hierarchy. You can operate on hierarchical designs from any level in the design without using the `current_design` command. For example, you can enter the following command to create a cell called foo in the design mid1:

```
dc_shell> create_cell mid1/foo my_lib/AND2
```

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, this sequence of commands replaces the reference for cell U8 with a high-power inverter.

```
dc_shell> get_pins U8/*
{"U8/A", "U8/Z"}
dc_shell> all_connected U8/A
{"n66"}
dc_shell> all_connected U8/Z
{"OUTBUS[10]"}
dc_shell> remove_cell U8
Removing cell 'U8' in design 'top'.
1
dc_shell> create_cell U8 IVP
Creating cell 'U8' in design 'top'.
1
dc_shell> connect_net n66 [get_pins U8/A]
Connecting net 'n66' to pin 'U8/A'.
1
dc_shell> connect_net OUTBUS[10] [get_pins U8/Z]
Connecting net 'OUTBUS[10]' to pin 'U8/Z'.
1
```

Note:

You can achieve the same result by using the `change_link` command instead of the series of commands listed above. For example, the following command replaces the reference for cell U8 with a high-power inverter:

```
dc_shell> change_link U8 IVP
```

Additional netlist editing commands similar to IC Compiler are available. These commands are described below:

- Resizing a cell

  You can use the `get_alternative_lib_cell` command to return a collection of equivalent library cells for a specific cell or library cell. You can then use the collection to replace or resize the cell. The `size_cell` command allows you to change the drive strength of a leaf cell by linking it to a new library cell that has the required properties.

- Inserting buffers or inverter pairs

  You can use the `insert_buffer` command to add a buffer at pins or ports. The `-inverter_pair` option allows you specify that a pair of inverting library cells is to be inserted instead of a single non-inverting library cell. To retrieve a collection of all buffers and inverters from the library, you can use the `get_buffer` command.

- Inserting repeaters

  The `-no_of_cells` option of the `insert_buffer` command allows you to select a driver of a two-pin net and insert a chain of single-fanout buffers in the net driven by this driver.

- Removing buffers

  You can use the `remove_buffer` command to remove buffers.

Translating Designs From One Technology to Another

You use the `translate` command to translate 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.

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.ddc
   ```

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

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

3. Invoke the `translate` command.

   ```
   dc_shell> translate
   ```

After a design is translated, you can compile it 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. For example,

```
dc_shell> set PORTS [all_inputs]
{"A0", "A1", "A2", "A3"}
dc_shell> query_objects $PORTS
PORTS = {"A0", "A1", "A2", "A3"}
dc_shell> remove_design
Removing design 'top'
1
dc_shell> 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 5-13 lists the design file formats supported by Design Compiler.

Table 5-13 Supported Output Formats

Format	Description
.ddc	Synopsys internal database format
Milkyway	Format for writing a Milkyway database within Design Compiler
Verilog	IEEE Standard Verilog (see the HDL Compiler documentation)
svsim	SystemVerilog netlist wrapper. Note: The `write -f svsim` command writes out only the netlist wrapper, not the gate-level DUT itself. To write out the gate-level DUT, you must use the existing `write -f verilog` command. For details, see the *SystemVerilog User Guide*.
VHDL	IEEE Standard VHDL (see the HDL Compiler documentation)

Commands to Save Design Files

Design Compiler provides the following ways to save design files:

- The `write` command
- The `write_milkyway` command

Using the write Command

You use the `write` command to convert designs in memory to a format you specify and save that representation to disk.

Use options to the `write` command as shown in Table 5-14.

Table 5-14 Using the write Command Options

To do this	Use this
Specify a list of designs to save. The default is the current design.	`design_list`
Specify the format in which a design is saved.	`-format` You can specify any of the output formats listed in Table 5-13 (except the Milkyway format; use the `write_milkyway` command instead)
Specify that all designs in the hierarchy are saved	`-hierarchy`
Specify a single file into which designs are written	`-output`
Specify that only modified designs are saved	`-modified`
Specify the name of the library in which the design is saved	`-library`

Using the write_milkyway Command

You use the `write_milkyway` command within dc_shell to write to a Milkyway database. The `write_milkyway` command creates a design file based on the netlist in memory and saves the design data for the current design in that file. For more information, see Chapter 11, "Using a Milkyway Database."

Saving Designs in .ddc Format

To save the design data in a .ddc file, use the `write -format ddc` command.

By default, the `write` command saves just the top-level design. To save the entire design, specify the `-hierarchy` option. If you do not use the `-output` option to specify the output file name, the `write -format ddc` command creates a file called *top_design*.ddc, where *top_design* is the name of the current design.

Example 1

The following command writes out all designs in the hierarchy of the specified design:

```
dc_shell> write -hierarchy -format ddc top
```

```
Writing ddc file 'top.ddc'
Writing ddc file 'A.ddc'
Writing ddc file 'B.ddc'
```

Example 2

The following command writes out multiple designs to a single file:

```
dc_shell> write -format ddc -output test.ddc {ADDER MULT16}

Writing ddc file 'test.ddc'
```

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

Example 5-1 shows sample naming rules as created by a specific layout tool vendor. These naming rules do the following:

- 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.

Example 5-1 Naming Rules Section of .synopsys_dc.setup File

```
define_name_rules simple_names -allowed "A-Za-z0-9_" \
-last_restricted "_" \
-first_restricted "_" \
-map { {{"\*cell\*","U"}, {"*-return","RET"}} }
```

Using the define_name_rules -map Command

Example 5-2 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 5-2 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 database 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 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 database 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 database file is not Verilog or VHDL compliant. For example, enter

   ```
   change_names -rules verilog -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 database file. Enter the following commands and compare the output.

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

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 5-37.

Summary of Commands for Changing Names

Table 5-15 summarizes commands for changing names.

Table 5-15 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.	change_names
Show effects of change_names without making the changes.	report_names
Define a set of rules for naming design objects. Name rules are used by change_names and report_names.	define_name_rules
List available name rules.	report_name_rules

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 5-16.

Table 5-16 Commands to Get Attribute Descriptions

Object type	Command
All	`man attributes`
Designs	`man design_attributes`
Cells	`man cell_attributes`
Clocks	`man clock_attributes`
Nets	`man net_attributes`
Pins	`man pin_attributes`

Table 5-16 Commands to Get Attribute Descriptions (Continued)

Object type	Command
Ports	`man port_attributes`
Libraries	`man library_attributes`
Library cells	`man library_cell_attributes`
References	`man reference_attributes`

Setting Attribute Values

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

- An attribute-specific command
- The `set_attribute` command

Using 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
```

Using 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 `dont_touch` attribute on the lsi_10k/FJK3 library cell, enter

```
dc_shell> set_attribute lsi_10K/FJK3 dont_use true
```

The `set_attribute` command enforces the predefined attribute type and generates an error if you try to set an attribute with a value of an incorrect type.

To determine the predefined type for an attribute, use the `list_attributes -application` command. This command generates a list of all application attributes and their types. To generate a smaller report, you can use the `-class` attribute to limit the list to attributes that apply to one of the following classes: design, port, cell, clock, pin, net, lib, or reference.

For example, the max_fanout attribute has a predefined type of float. Suppose you enter the following command, Design Compiler displays an error message:

```
set_attribute lib/lcell/lpin max_fanout 1 -type integer
```

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 5-43.

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 -obj_type object
```

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 5-43.

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 "Using the set_attribute Command" on page 5-41.

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 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.

You can override the default search order by using the dctcl 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
1
```

To place the dont_touch attribute on the net instead, use the following command:

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

6

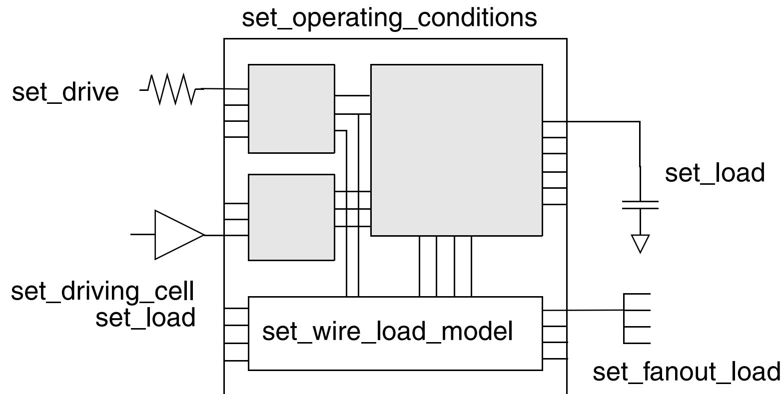
Defining the Design Environment

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
- Setting Logic Constraints on Ports
- Specifying Power Intent
- Support for Multicorner-Multimode Designs

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_file 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: X-2005.09
Date   : Mon Jan 13 10:56:49 2005
```

```
****************************************
...
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 models are used only when Design Compiler is not operating in topographical mode. For topographical mode details, see Chapter 10, "Using Design Compiler Topographical Technology." 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.

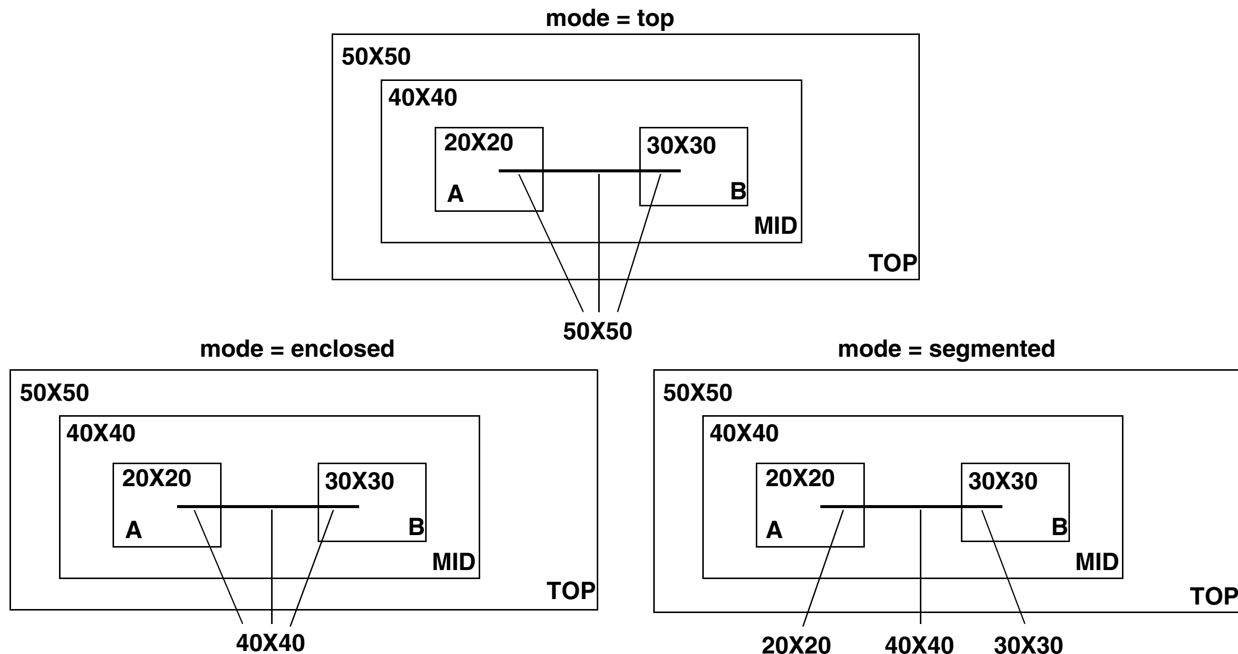
- Segmented

  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_file 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: Y-2006.06
Date   : Mon May 1 10:56:49 2006
****************************************
...
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

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. The wire load model selected the first time you compile is used in subsequent compiles.

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 the following commands:

```
dc_shell> 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 design rule constraints.

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.

Use the `remove_driving_cell` command or `reset_design` command to remove driving cell attributes on ports.

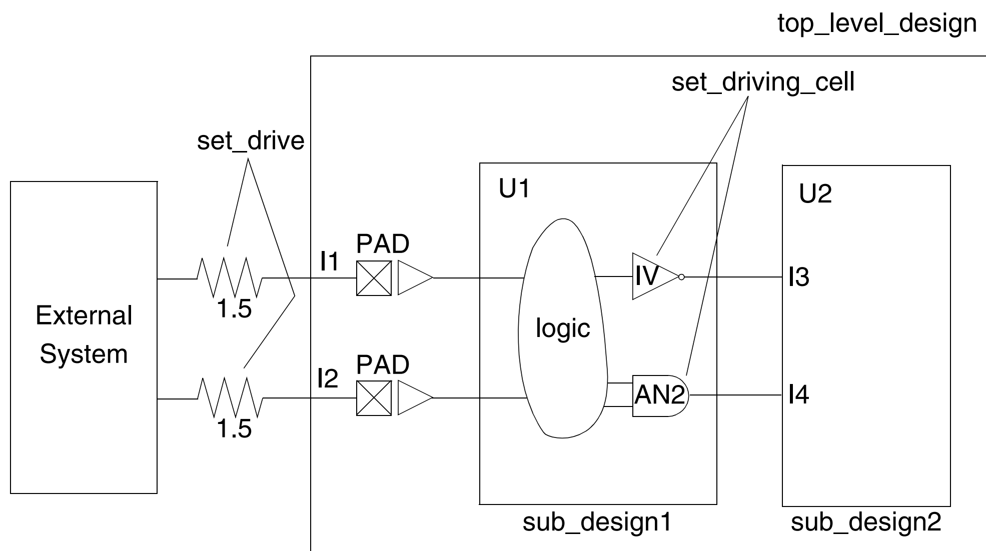
The set_drive and set_input_transition Commands

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 -lib_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 -lib_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 "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.

Setting Logic Constraints on Ports

Design Compiler provides commands for setting ports to improve optimization quality. Table 6-1 lists the commands that eliminate redundant ports or inverters.

Table 6-1 Commands Eliminating Redundant Ports or Inverters

Command	Description
`set_equal`	Defines ports as logically equivalent
`set_opposite`	Defines ports as logically opposite
`set_logic_dc`	Specifies one or more ports driven by don't care
`set_logic_one`	Specifies one or more ports tied to logic 1
`set_logic_zero`	Specifies one or more ports tied to logic 0
`set_unconnected`	Lists output ports to be unconnected

The following sections describe these commands in detail.

Defining Ports as Logically Equivalent

Some input ports are driven by logically related signals. For example, the signals driving a pair of input ports might always be the same (logically equal) or might always be different (logically opposite).

The `set_equal` command specifies that two input ports are logically equivalent.

The syntax is

```
set_equal  port1 port2
```

Note:
For more information, see the `set_equal` man page.

Example

To specify that ports IN_X and IN_Y are equal, enter

```
dc_shell> set_equal IN_X IN_Y
```

To remove this attribute, use the `reset_design` command. The `reset_design` command removes all user-specified objects and attributes from the current design, except those defined with the `set_attribute` command.

Defining Logically Opposite Input Ports

The `set_opposite` command specifies that two input ports in the current designs are logically opposite.

Use `set_opposite` to eliminate redundant inverters and to improve the quality of optimization.

The syntax is

```
set_opposite  port1 port2
```

Note:
For more information, see the `set_opposite` man page.

To remove this attribute, use the `reset_design` command, which removes all user-specified objects and attributes from the current design, except those defined with the `set_attribute` command. To remove those defined with `set_attribute`, use the `remove_attribute` command.

Allowing Assignment of Any Signal to an Input

The `set_logic_dc` command specifies an input port driven by don't care.

This information is used to create smaller designs during compile. After optimization, a port connected to don't care usually does not drive anything inside the optimized design.

Use `set_logic_dc` to allow assignment of any signal to that input (including but not limited to 0 and 1 during compilation). The outputs of the design are significant only when the inputs that are not don't care completely determine all the outputs, independent of the don't care inputs.

Use `set_logic_dc` on input ports. To specify output ports as unused, use the `set_unconnected` command.

The syntax is

```
set_logic_dc  port_list
```

Note:
For more information, see the `set_logic_dc` man page.

Example 1

```
dc_shell> set_logic_dc { A B }
```

For a 2:1 multiplexer design with inputs S, A, B, and output Z, the function is computed as

```
Z = S*A + S'*B
```

The command `set_logic_dc` B implies that the value of Z is significant when S = 1 and is a don't care when S = 0. The resulting simplification, done during compilation, gives the reduced logic as a wire and the function is

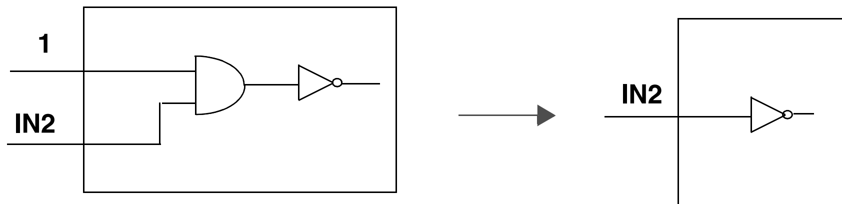
```
Z = A
```

To undo this command, use the `remove_attribute` command.

Specifying Input Ports Always One or Zero

If an input port is always logic-high or -low, Design Compiler might be able to simplify the surrounding logic function during optimization and create a smaller design. Figure 6-4 shows simplified input port logic.

Figure 6-4 Simplified Input Port Logic



You can specify that input ports are connected to logic 1 or logic 0.

Tying Input Ports to Logic 1

The `set_logic_one` command lists the input ports tied to logic 1.

After optimization, a port connected to logic 1 usually does not drive anything inside the optimized design.

Use `set_logic_one` on input ports. To specify output ports as unused, use the `set_unconnected` command.

The syntax is

```
set_logic_one port_list
```

Note:
For more information, see the `set_logic_one` man page.

To undo this command, use the `remove_attribute` command.

Tying Input Ports to Logic 0

The `set_logic_zero` command lists input ports tied to logic 0.

After optimization, a port connected to logic 0 usually does not drive anything inside the optimized design.

Use `set_logic_zero` on input ports. To specify output ports as unused, use the `set_unconnected` command.

The syntax is

```
set_logic_zero  port_list
```

Note:
For more information, see the `set_logic_zero` man page.

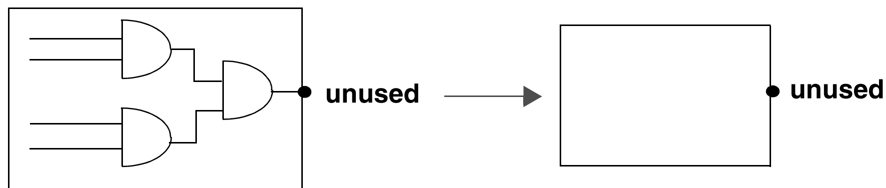
Example

```
dc_shell> set_logic_one IN
```

Specifying Unconnected Output Ports

If an output port is not used, that is, if it is unconnected, the logic driving the port can be minimized or eliminated during optimization. Figure 6-5 shows an example.

Figure 6-5 Minimizing Logic Driving an Unconnected Output Port



The `set_unconnected` command specifies output ports to be unconnected to outside logic.

The syntax is

```
set_unconnected  port_list
```

Note:
For more information, see the `set_unconnected` man page.

Example

```
dc_shell> set_unconnected OUT
```

To undo this command, use the `remove_attribute` command.

Specifying Power Intent

The IEEE™ 1801 Unified Power Format (UPF) Standard establishes set of commands used to specify the low-power design intent for electronic systems. Using UPF commands, you can specify the supply network, switches, isolation, retention, and other aspects relevant to power management of a chip design. The same set of low-power design specification commands is to be used throughout the design, analysis, verification, and implementation flow. Synopsys tools are designed to follow the official UPF standard.

Power Intent Concepts

The IEEE 1801™ (UPF) language provides a way to specify the power requirements of a design, but without specifying explicitly how those requirements are implemented. The language specifies how to create a power supply network for each design element, the behavior of supply nets with respect to each other, and how the logic functionality is extended to support dynamic power switching to design elements. It does not contain any placement or routing information. The UPF specification is separate from the RTL description of the design.

In the UPF language, a *power domain* is a defined group of elements in the logic hierarchy that share a common set of power supply needs. By default, all logic elements in a power domain use the same primary supply and primary ground. Other power supplies may optionally be defined for a power domain as well. A power domain is typically implemented as a contiguous *voltage area* in the physical chip layout, although this is not a requirement of the language.

Each power domain has a *scope* and an *extent*. The *scope* is the level of logic hierarchy where the power domain exists. The *extent* is the set of logic elements that belong to the power domain and share the same power supply needs. In other words, the scope is the hierarchical level where the power domain exists, whereas the extent is what is contained within the power domain.

Each scope or hierarchical level in the design has *supply nets* and *supply ports*. A *supply net* is a conductor that carries a supply voltage or ground throughout a given power domain. A supply net that spans more than one power domain is said to be “reused” in multiple domains. A *supply port* is a power supply connection point between two adjacent levels of the design hierarchy, between parent and child blocks of the hierarchy. A supply net that crosses from one level of the design hierarchy to the next must pass through a supply port.

A *power switch* (or simply *switch*) is a device that turns on and turns off power for a supply net. A switch has an in input supply net, an output supply net that can be switched on or off, and at least one input signal to control switching. The switch can optionally have multiple input control signals and one or more output acknowledge signals. A *power state table* lists the allowed combinations of voltage values and states of the power switches for all power domains in the design.

Where a logic signal leaves one power domain and enters another at a substantially different supply voltage, a *level-shifter* cell must be present to convert the signal from the voltage swing of the first domain to that of the second domain.

Where a logic signal leaves a power domain and enters a different power domain, an *isolation* cell must be present to generate a known logic value during shutdown. If the voltage levels of the two domains are substantially different, the interface cell must perform both level shifting when the domain is powered up and isolation when the domain is powered down. A cell that can perform both functions is called an *enable level shifter*.

In a power domain that has power switching, any registers that are to retain data during shutdown must be implemented as *retention registers*. A retention register has a separate, always-on supply net, sometimes called the backup supply, which keeps the data stable in while the primary supply of the domain is shut down.

UPF Commands in Synopsys Tools

In addition to supporting most of the commands in the UPF standard, Design Compiler supports non-UPF commands that report power-related objects in the design, such as the `report_power_domain`, `report_power_switch`, `report_supply_net`, and `report_supply_port` reports. Each command reports the objects existing in the design and the characteristics of the object. For example, the `report_supply_net` command lists the supply nets and shows the name, scope, power domains, supply ports, power pins, voltage, and resolution status of each supply net. Other commands are available that create collections of such objects, such as `get_power_domains`, `get_power_switches`, `get_supply_nets`, and `get_supply_ports`. The reporting and collection commands are not part of the UPF specification.

For more information about a particular reporting or collection command, see its man page in the applicable tool.

Table 6-2 lists the major UPF commands and shows whether they are supported by Design Compiler. A “yes” entry in the table indicates support for the UPF commands listed in the left column. A blank space indicates that the commands do not apply to Design Compiler.

Table 6-2 Support for IEEE 1801 (UPF) Commands in Design Compiler

IEEE 1801 (UPF) commands	Design Compiler support
Load/save/scope:	
upf_version	--
set_scope	yes
load_upf (partial)	yes
save_upf (partial)	yes

Table 6-2 Support for IEEE 1801 (UPF) Commands in Design Compiler (Continued)

IEEE 1801 (UPF) commands	Design Compiler support
Basic power:	
create_power_domain	yes
create_supply_net	yes
create_supply_port	yes
set_domain_supply_net	yes
connect_supply_net (partial)	yes
create_power_switch (partial)	yes
map_power_switch	--
Level shifter:	
set_level_shifter	yes
map_level_shifter_cell	yes
name_format	yes
Isolation:	
set_isolation	yes
set_isolation_control	yes
map_isolation_cell	yes
name_format	yes
Retention:	
set_retention	yes
set_retention_control	yes
map_retention_cell (partial)	yes
Power state table:	
add_port_state	yes
add_pst_state	yes
create_pst	yes
Verification extension:	
set_design_top	--

The commands to load and execute UPF commands from a file, to write a set of UPF commands to a file, and to set the hierarchical scope for subsequent UPF commands are described in the following sections.

upf_version

```
upf_version string
```

The `upf_version` command specifies the UPF version number for which the subsequent UPF syntax is intended, or it returns the current UPF version number. As of the current release, only version 1.0 is supported.

set_scope

```
set_scope [instance]
```

The `set_scope` command specifies the hierarchical scope of subsequent commands, including UPF commands. It has the same effect as the `current_instance` command. If no instance name is specified, the scope is set to the top level of the design. If the instance is specified as a single period character (.), the scope remains at the current instance. If the instance is specified as two period characters (..), the context is moved up one level in the instance hierarchy. If the instance string begins with a slash character (/), the scope changes to the instance whose name follows the slash character, relative to the top of the design.

The `set_scope` command reports the scope setting (a hierarchical path) before the setting is changed. This is different from the `current_instance` command, which reports the scope setting after the setting has been changed.

load_upf

```
load_upf upf_file_name
 [-scope instance_name]
```

The `load_upf` command executes the UPF commands in a specified file. The commands describe the power intent of the design in the same way that the commands read by the `read_sdc` command describe the timing constraints on a design.

You can optionally specify a scope for the command. The scope is a name of a hierarchical instance on which the commands are applied. Specifying a scope has the same effect as changing the tool to the level of instance with the `set_scope` or `current_instance` command, executing the commands, and then changing back to the original hierarchical level. If you do not specify a scope, the commands are applied to the current scope.

Design Compiler lets you remove loaded UPF commands with the `remove_upf` command.

save_upf

```
save_upf upf_file_name
```

The `save_upf` command writes out a set of UPF commands to a specified file. The commands fully describe the power intent of the design in the same way that the commands written by the `write_sdc` command describe the timing constraints on a design.

For more information on UPF commands, see the *Synopsys Low-Power Flow User Guide*.

Support for Multicorner-Multimode Designs

Designs are often required to operate under multiple modes, such as test or standby mode, and multiple operating conditions, sometimes referred to as corners. Such designs are known as multicorner-multimode designs. Design Compiler Graphical can analyze and optimize across multiple modes and corners concurrently. The multicorner-multimode feature in Design Compiler Graphical provides compatibility between flows in Design Compiler and IC Compiler.

To define your modes and corners, you use the `create_scenario` command. A scenario definition usually includes commands that specify the TLUPlus libraries, operating conditions, and constraints. For details on setting up multicorner-multimode analysis, see "Optimizing Multicorner-Multimode Designs in Design Compiler Graphical" on page 10-102.

7

Defining Design Constraints

Constraints are declarations that define your design's goals in measurable circuit characteristics such as timing, area, and capacitance. Without constraints, the Design Compiler tool cannot effectively optimize your design.

In this chapter, you will learn about the following:

- Design Compiler Constraint Types
- Design Rule Constraints
- Optimization Constraints
- Managing Constraint Priorities
- Reporting Constraints
- Propagating Constraints in Hierarchical Designs

Design Compiler Constraint Types

When Design Compiler optimizes your design, it uses two types of constraints:

Design rule constraints

These are implicit constraints; the technology library defines them. These constraints are requirements for a design to function correctly, and they apply to any design using the library. You can make these constraints more restrictive than optimization constraints.

Optimization constraints

These are explicit constraints; you define them. Optimization constraints apply to the design on which you are working for the duration of the dc_shell session and represent the design's goals. They must be realistic.

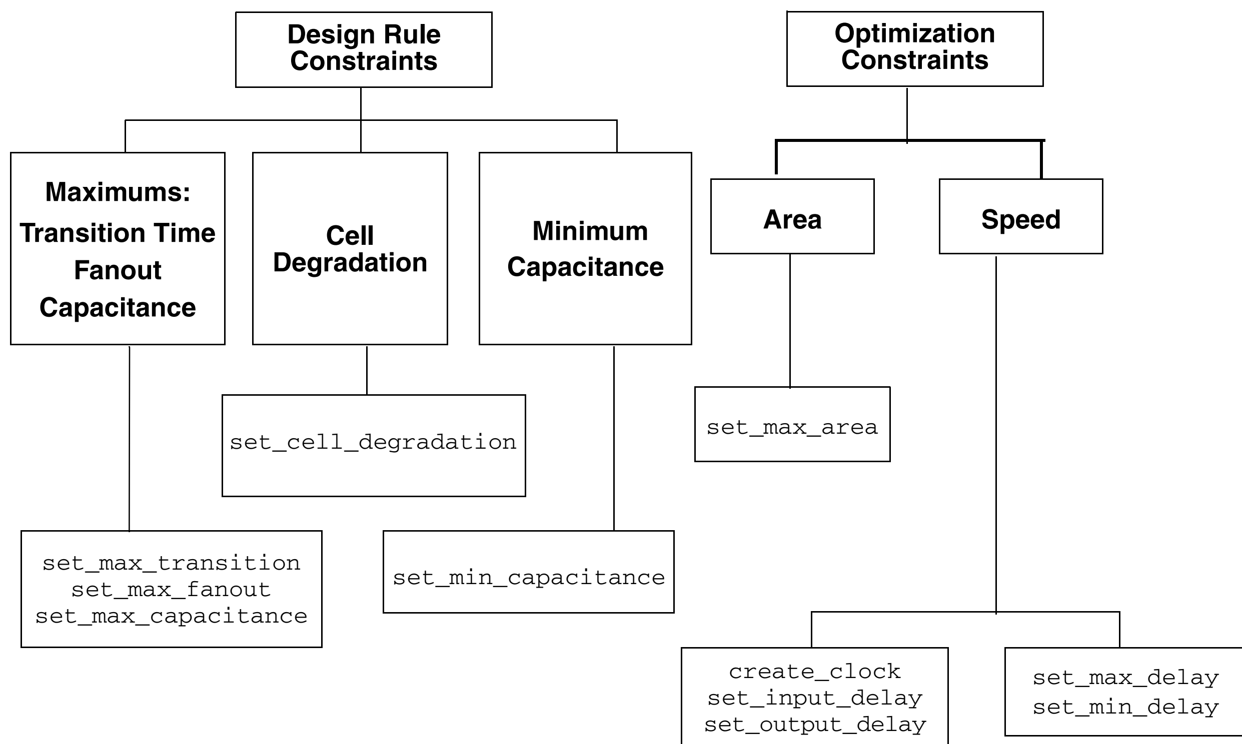
Design Compiler tries to meet both design rule constraints and optimization constraints, but design rule constraints take precedence.

Design Compiler calculates two cost functions: one for design rule constraints and one for optimization constraints. During gate-level optimization, Design Compiler reports the value of each cost function whenever a change is made to the design.

You specify constraints interactively on the command line or in a constraints file.

Figure 7-1 shows the major Design Compiler design rule constraints and optimization constraints and the dc_shell interface commands to set the constraints.

Figure 7-1 Major Design Compiler Constraints



Note:
If you use the Power Compiler tool from Synopsys, `max_dynamic_power` and `max_leakage_power` are optimization constraints. To use these constraints, you need a Power Optimization license and supporting libraries characterized for power.

Design Rule Constraints

Design rule constraints reflect technology-specific restrictions your design must meet in order to function as intended.

Design rules constrain the nets of a design but are associated with the pins of cells from a technology library. Most technology libraries specify default design rules. Design Compiler cannot violate design rule constraints, even if it means violating optimization constraints (delay and area goals). You can apply more restrictive design rules, but you cannot apply less restrictive ones.

The design rule constraints comprise

- Maximum transition time
- Maximum fanout
- Minimum and maximum capacitance
- Cell degradation

You cannot remove max_transition, max_fanout, max_capacitance, and min_capacitance attributes set in a technology library, because they are requirements for the technology, but you can set values that are more restrictive. If both implicit and explicit values are set on a design or a port, the more restrictive value applies. You can remove values you have set.

Design Rule Cost Function

Design Compiler helps you fix design rule violations. If there are multiple violations, Design Compiler tries to fix the violation with the highest priority first. When possible, it also evaluates and selects alternatives that reduce violations of other design rules. Design Compiler uses the same approach with delay violations.

Figure 7-2 shows the design rule cost function equation.

Figure 7-2 Design Rule Cost Equation

$$\sum_{i=1}^{m} max(d_i,\ 0) \times w_i$$

i = Index
d = Delta Constraint
m = Total Number of Constraints
w = Constraint Weight

In a compilation report, the DESIGN RULE COST field reports the cost function for the design rule constraints on the design.

Maximum Transition Time

Maximum transition time is a design rule constraint.

The maximum transition time for a net is the longest time required for its driving pin to change logic values. Many technology libraries contain restrictions on the maximum transition time for a pin, creating an implicit transition time limit for designs using that library.

Design Compiler attempts to make the transition time of each net less than the max_transition value, for example, by buffering the output of the driving gate. The max_transition value can vary with the operating frequency of a cell.

Transition times on nets are computed by use of timing data from the technology library.

To change or add to the implicit transition time values from a technology library, use the set_max_transition command.

If both a library max_transition and a design max_transition attribute are defined, Design Compiler tries to meet the smaller (more restrictive) value.

If your design uses multiple technology libraries and each has a different default max_transition value, Design Compiler uses the smallest max_transition value globally across the design.

Defining Maximum Transition Time

The set_max_transition command sets the max_transition attribute to the specified value on clock groups, ports, or designs. During compile, Design Compiler attempts to ensure that the transition time for a net is less than the specified value.

Example

To set a maximum transition time of 3.2 for the design adder, enter the following command:

```
dc_shell> set_max_transition 3.2 [get_designs adder]
```

To undo a set_max_transition command, use remove_attribute. Enter the following command:

```
dc_shell> remove_attribute [get_designs adder] max_transition
```

For command details, see the man page.

Specifying Clock-Based Maximum Transition

The max_transition value can vary with the operating frequency of a cell. The operating frequency of a cell is defined as the highest clock frequency on the registers driving a cone of logic (clock frequencies are defined with the create_clock command).

For designs with multiple clock domains, you can use the set_max_transition command to set the max_transition attribute on pins in a specific clock group.

Design Compiler follows these rules in determining the max_transition value:

- When the max_transition attribute is set on a design or port and a clock group, the most restrictive constraint is used.
- If multiple clocks launch the same paths, the most restrictive constraint is used.

- If max_transition attributes are already specified in a technology library, the tool automatically attempts to meet these constraints during compile.

For example the following command sets a max_transition value of 5 on all pins belonging to the Clk clock group:

```
dc_shell> set_max_transition 5 [get_clocks Clk]
```

Maximum Fanout

Maximum fanout is a design rule constraint.

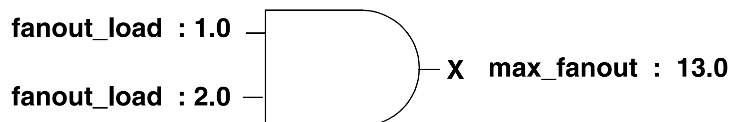
Most technology libraries place fanout restrictions on driving pins, creating an implicit fanout constraint for every driving pin in designs using that library.

You can set a more conservative fanout constraint on an entire library or define fanout constraints for specific pins in the library description of an individual cell.

If a library fanout constraint exists and you specify a max_fanout attribute, Design Compiler tries to meet the smaller (more restrictive) 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. Figure 7-3 shows these fanout attributes.

Figure 7-3 fanout_load and max_fanout Attributes



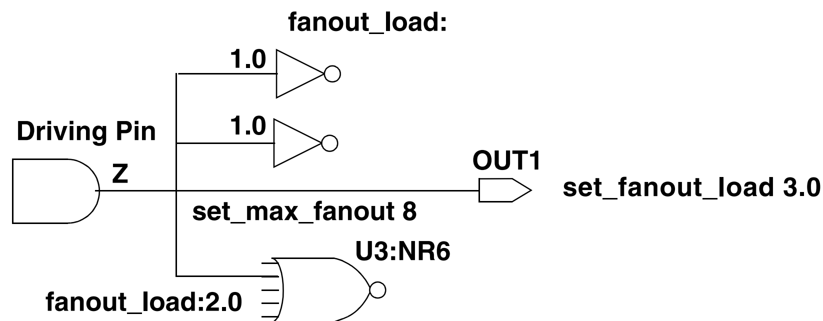
To evaluate the fanout for a driving pin (such as X in Figure 7-3), Design Compiler calculates the sum of all the fanout_load attributes for inputs driven by pin X and compares that number with the number of max_fanout attributes stored at the driving pin X.

- If the sum of the fanout loads is not more than the max_fanout value, the net driven by X is valid.
- If the net driven by X is not valid, Design Compiler tries to make that net valid, perhaps by choosing a higher-drive component.

Maximum Fanout Calculation Example

Figure 7-4 shows how maximum fanout is calculated.

Figure 7-4 Calculation of Maximum Fanout



You can set a maximum fanout constraint on every driving pin and input port as follows:

```
dc_shell> set_max_fanout 8 [get_designs ADDER]
```

To check whether the maximum fanout constraint is met for driving pin Z, Design Compiler compares the specified `max_fanout` attribute against the fanout load.

In this case, the design meets the constraints.

Total Fanout Load

$$8 \geq \underbrace{1.0 + 1.0 + 3.0 + 2.0}_{7}$$

The fanout load imposed by a driven cell (U3) is not necessarily 1.0. Library developers can assign higher fanout loads (for example, 2.0) to model internal cell fanout effects.

You can also set a fanout load on an output port (OUT1) to model external fanout effects.

Fanout load is a dimensionless number, not a capacitance. It represents a numerical contribution to the total effective fanout.

Defining Maximum Fanout

The `set_max_fanout` command sets the maximum allowable fanout load for the listed input ports. The `set_max_fanout` command sets a `max_fanout` attribute on the listed objects.

To undo a maximum fanout value set on an input port or design, use the `remove_attribute` command. For example,

```
dc_shell> remove_attribute [get_ports port_name] max_fanout
dc_shell> remove_attribute [get_designs design_name]max_fanout
```

For more information, see the man page.

Defining Expected Fanout for Output Ports

The `set_fanout_load` command sets the expected fanout load value for listed output ports.

Design Compiler adds the fanout value to all other loads on the pin driving each port in *port_list* and tries to make the total load less than the maximum fanout load of the pin.

To undo a `set_fanout_load` command set on an output port, use the `remove_attribute` command. For example,

```
dc_shell> remove_attribute port_name fanout_load
```

To determine the fanout load, use the `get_attribute` command.

Examples

To find the fanout load on the input pin of library cell AND2 in library libA, enter

```
dc_shell> get_attribute "libA/AND2/i" fanout_load
```

To find the default fanout load set on technology library libA, enter

```
dc_shell> get_attribute libA default_fanout_load
```

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

Maximum Capacitance

Maximum capacitance is a design rule constraint. It is set as a pin-level attribute that defines the maximum total capacitive load that an output pin can drive. That is, the pin cannot connect to a net that has a total capacitance (load pin capacitance and interconnect capacitance) greater than or equal to the maximum capacitance defined at the pin.

The maximum capacitance design rule constraint allows you to control the capacitance of nets directly. (The design rule constraints `max_fanout` and `max_transition` limit the actual capacitance of nets indirectly.) The `max_capacitance` constraint operates similarly to `max_transition`, but the cost is based on the total capacitance of the net, rather than the transition time. The `max_capacitance` attribute functions independently, so you can use it with `max_fanout` and `max_transition`.

The `max_capacitance` value can vary with the operating frequency of a cell. You can have Design Compiler annotate each driver pin with a frequency-based `max_capacitance` value by setting the `compile_enable_dyn_max_cap` variable to true. See “Specifying Frequency-Based Maximum Capacitance” on page 7-9.

The `max_capacitance` constraint has priority over the cell degradation constraint.

If both a library max_capacitance attribute and a design max_capacitance attribute exist, Design Compiler tries to meet the smaller (more restrictive) value.

Defining Maximum Capacitance

The set_max_capacitance command sets a maximum capacitance for the nets attached to named ports or to all the nets in a design. This command allows you to control capacitance directly and places a max_capacitance attribute on the listed objects.

Design Compiler calculates the capacitance on a net by adding the wire capacitance of the net to the capacitance of the pins attached to the net. To determine whether a net meets the capacitance constraint, Design Compiler compares the calculated capacitance value with the max_capacitance value of the pin driving the net.

Use the set_max_capacitance command to specify a capacitance value on input ports or designs. This value should be less than or equal to the max_capacitance of the pin driving the net.

Examples

To set a maximum capacitance of 3 for the design adder, enter one the following command:

```
dc_shell> set_max_capacitance 3 [get_designs adder]
```

To undo a set_max_capacitance command, use remove_attribute. For example, enter the following command:

```
dc_shell> remove_attribute [get_designs adder] max_capicitance
```

For more information, see the set_max_capacitance man page.

Specifying Frequency-Based Maximum Capacitance

The max_capacitance value can vary with the operating frequency of a cell. You can have Design Compiler annotate each driver pin with a frequency-based max_capacitance value by setting the compile_enable_dyn_max_cap variable to true.

Before you use this feature, your technology library should be characterized for multiple frequencies. This characterization consists of associating a max_capacitance value with each driver pin for each frequency and capturing this information in a one-dimensional lookup table. For information on creating the max_capacitance lookup table, see the Library Compiler documentation.

Design Compiler follows these steps in determining the max_capacitance value:

1. It identifies the operating frequency of each cell.

   The operating frequency of a cell is defined as the highest clock frequency on the registers driving a cone of logic (clock frequencies are defined with the create_clock command).
2. It looks up the corresponding max_capacitance value for the identified frequency from the lookup table in the technology library.
3. It annotates each driver pin with the appropriate max_capacitance value and uses these values in synthesis.

Minimum Capacitance

Minimum capacitance is a design rule constraint. Some technology libraries specify minimum capacitance.

The min_capacitance design rule specifies the minimum load a cell can drive. It specifies the lower bound of the range of loads with which a cell has been characterized to operate. During optimization, Design Compiler ensures that the load driven by a cell meets the minimum capacitance requirement for that cell. When a violation occurs, Design Compiler fixes the violation by sizing the driver.

You can use the min_capacitance design rule with the existing design rules. The min_capacitance has higher priority than the maximum transition time, maximum fanout, and maximum capacitance constraints.

You can set a minimum capacitance for nets attached to input ports to determine whether violations occur for driving cells at the input boundary. If violations are reported after compilation, you can fix the problem by recompiling the module driving the ports. Use set_min_capacitance for input or inout ports; you cannot set minimum capacitance on a design.

If library min_capacitance and design min_capacitance attributes both exist, Design Compiler tries to meet the larger (more restrictive) value.

Defining Minimum Capacitance

The set_min_capacitance command sets a defined minimum capacitance value on listed input or bidirectional ports.

To set a minimum capacitance for nets attached to input or bidirectional ports, use the set_min_capacitance command.

Example

To set a minimum capacitance value of 12.0 units on the port named high_drive, enter

```
dc_shell> set_min_capacitance 12.0 high_drive
```

To report only minimum capacitance constraint information, use the `-min_capacitance` option of the `report_constraint` command.

To get information about the current port settings, use the `report_port` command.

To undo a `set_min_capacitance` command, use the `remove_attribute` command.

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

Cell Degradation

Cell degradation is a design rule constraint. Some technology libraries contain cell degradation tables. The tables list the maximum capacitance that can be driven by a cell as a function of the transition times at the inputs of the cell.

The cell_degradation design rule specifies that the capacitance value for a net is less than the cell degradation value.

The `cell_degradation` design rule can be used with other design rules, but the `max_capacitance` design rule has a higher priority than the `cell_degradation` design rule. If the `max_capacitance` rule is not violated, applying the `cell_degradation` design rule does not cause it to be violated.

The `set_cell_degradation` command sets the `cell_degradation` attribute to a specified value on specified input ports.

During compilation, if `cell_degradation` tables are specified in a technology library, Design Compiler tries to ensure that the capacitance value for a net is less than the specified value. The `cell_degradation` tables give the maximum capacitance that a cell can drive, as a function of the transition times at the inputs of the cell.

If cell_degradation tables are not specified in a technology library, you can set `cell_degradation` explicitly on the input ports.

By default, a port has no `cell_degradation` constraint.

Note:
  Use of the `set_cell_degradation` command requires a DC Ultra license.

The syntax is

```
set_cell_degradation  cell_deg_value object_list
```

Note:
For more information, see the `set_cell_degradation` man page.

Example

This command sets a maximum capacitance value of 2.0 units on the port named late_riser:

```
dc_shell> set_cell_degradation 2.0 late_riser
```

To get information about optimization and design rule constraints, use the `report_constraint` command.

To remove the `cell_degradation` attribute, use the `remove_attribute` command.

Connection Class

The connection class constraint on a port describes the connection requirements for a given technology. Only loads and drivers with the same connection class label can be legally connected. This constraint can be specified in the library or by the user. To set a connection class constraint on a port, use the `set_connection_class` command. For details, see the man page for the command.

Managing Design Rule Constraint Priorities

By default, design rule constraints have a higher optimization priority than optimization constraints. You can change that priority, however; see "Managing Constraint Priorities" on page 7-19.

Precedence of Design Rule Constraints

Design Compiler follows this descending order of precedence when it tries to resolve conflicts among design rule constraints (see Table 7-2 on page 7-19):

1. Minimum capacitance
2. Maximum transition
3. Maximum fanout
4. Maximum capacitance
5. Cell degradation

The following details apply to the precedence of design rule constraints:

- Maximum transition has precedence over maximum fanout. If a maximum fanout constraint is not met, investigate the possibility of a conflicting maximum transition constraint. Design Compiler does not make a transition time worse in order to fix a maximum fanout violation.
- Maximum fanout has precedence over maximum capacitance.
- Design Compiler calculates transition time for a net in two ways, depending on the library.
  - For libraries using the CMOS delay model, Design Compiler calculates the transition time by using the drive resistance of the driving cell and the capacitive load on the net.
  - For libraries using a nonlinear delay model, Design Compiler calculates the transition time by using table lookup and interpolation. This is a function of capacitance at the output pin and of the input transition time.
- The `set_driving_cell` and `set_drive` commands behave differently, depending on your technology library.
  - For libraries using the CMOS delay model, drive resistance is a constant. In this case, the `set_drive -drive_of` command and the `set_driving_cell` command give the same result.
  - For libraries using a nonlinear delay model, the `set_driving_cell` command calculates the transition time dynamically, based on the load from the tables. The `set_drive` command returns one value when the command is issued and uses a value from the middle of the range in the tables for load.

Both the `set_driving_cell` and the `set_drive` commands affect the port transition delay. The `set_driving_cell` command places the design rule constraints, annotated with the driving cell, on the affected port.

The `set_load` command places a load on a port or a net. The units of this load must be consistent with your technology library. This value is used for timing optimizations, not for maximum fanout optimizations.

Design Rule Scenarios

Typical design rule scenarios are

- `set_max_fanout` and `set_max_transition` commands
- `set_max_fanout` and `set_max_capacitance` commands

Typically, a technology library specifies a default `max_transition` or `max_capacitance`, but not both. To achieve the best result, do not mix `max_transition` and `max_capacitance`.

Disabling Design Rule Fixing on Special Nets

You use the `set_auto_disable_drc_nets` command to enable or disable design rule fixing on clock, constant, or scan nets. The command acts on all the nets of a given type in the current design; that is, depending on the options you specify, it acts on all the clock nets, or all the constant nets, or all the scan nets, or on any combination of these three net types. Nets that are disabled with respect to design rule fixing are marked with the `auto_disable_drc_nets` attribute.

By default, the clock and constant nets of a design have design rule fixing disabled (this is the same as using the `-default` option of the command); the scan nets do not. You can use the `-all` option to disable design rule fixing on all three net types. Alternatively, you can independently disable or enable design rule fixing for the clock nets, constant nets, or scan nets by assigning a true or false value to the `-clock`, `-constant`, or `-scan` options, respectively. Finally, you can use the `-none` option to *enable* design rule fixing on all three types of nets.

Note:
Clock nets are ideal nets by default. Using the `set_auto_disable_drc_nets` command to *enable* design rule fixing does not affect the ideal timing properties of clock nets. You must use the `set_propagated_clock` command to affect the ideal timing of clock nets.

You cannot use the `set_auto_disable_drc_nets` to override disabled design rule fixing on ideal networks marked with ideal network attributes. This command never overrides the settings specified by the `set_ideal_net` or the `set_ideal_network` command.

Summary of Design Rule Commands and Objects

Table 7-1 summarizes the design rule commands and the objects on which to set them.

Table 7-1 Design Rule Command and Object Summary

Command	Object
`set_max_fanout`	Input ports or designs
`set_fanout_load`	Output ports
`set_load`	Ports or nets
`set_max_transition`	Ports or designs
`set_cell_degradation`	Input ports
`set_min_capacitance`	Input ports

Optimization Constraints

Optimization constraints represent speed and area design goals and restrictions that you want but that might not be crucial to the operation of a design. Speed (timing) constraints have higher priority than area.

By default, optimization constraints are secondary to design rule constraints. (That priority can be changed, however; see "Managing Constraint Priorities" on page 7-19.)

The optimization constraints comprise

- Timing constraints (performance and speed)
    - Input and output delays (synchronous paths)
    - Minimum and maximum delay (asynchronous paths)
- Maximum area (number of gates)

Optimization Cost Function

During the first phase of mapping, Design Compiler works to reduce the optimization cost function. Design Compiler evaluates this function to determine whether a change to the design improves the cost.

The full optimization cost function takes into account the following components, listed in order of importance. Not all components are active on all designs.

1. Maximum delay cost
2. Minimum delay cost
3. Maximum area cost

Design Compiler evaluates cost function components independently in order of importance and accepts an optimization move if it decreases the cost of one component without increasing more-important costs. For example, an optimization move that improves maximum delay cost is always accepted.

Optimization stops when all costs are zero or no further improvements can be made to the cost function.

Timing Constraints

When defining timing constraints you should consider that your design has synchronous paths and asynchronous paths.

Synchronous paths are constrained by specifying clocks in the design. Use the `create_clock` command to specify a clock. After specifying the clocks, it is recommended you also specify the input and output port timing specifications. Use the `set_input_delay` and `set_output_delay` commands.

Asynchronous paths are constrained by specifying minimum and maximum delay values. Use the `set_max_delay` and `set_min_delay` commands to specify these point-to-point delays. The calculation of minimum and maximum delays are described in the following sections.

For additional information, see the *Synopsys Timing Constraints and Optimization User Guide*.

Maximum Delay

Maximum delay is an optimization constraint.

Design Compiler contains a built-in static timing analyzer for evaluating timing constraints. A static timing analyzer calculates path delays from local gate and interconnect delays but does not simulate the design. The Design Compiler timing analyzer performs critical path tracing to check minimum and maximum delays for every timing path in the design. The most critical path is not necessarily the longest combinational path in a sequential design, because paths can be relative to different clocks at path startpoints and endpoints.

The timing analyzer calculates minimum and maximum signal rise and fall path values based on the timing values and environmental information in the technology library.

Cost Calculation

Maximum delay is usually the most important portion of the optimization cost function. The maximum delay optimization cost guides Design Compiler to produce a design that functions at the speed you want.

The maximum delay cost includes multiple parts. Figure 7-5 shows the maximum delay cost equation.

Figure 7-5 Cost Calculation for Maximum Delay

$$\sum_{i=1}^{m} v_i \times w_i$$

i = Index
v = worst violation
m = number of path groups
w = weight

Maximum delay target values for each timing path in the design are automatically determined after considering clock waveforms and skew, library setup times, external delays, multicycle or false path specifications, and `set_max_delay` commands. Load, drive, operating conditions, wire load model, and other factors are also taken into account.

The maximum delay cost is affected by how paths are grouped by the `group_path` and `create_clock` commands.

- If only one path group exists, the maximum delay cost is the cost for that group: the amount of the worst violation multiplied by the group weight.
- If multiple path groups exist, the costs for all the groups are added together to determine the maximum delay cost of the design. The group cost is always zero or greater.

```
Delta = max(delta(pin1), delta(pin2), ... delta(pinN))
```

Minimum Delay

Minimum delay is an optimization constraint, but Design Compiler fixes the minimum delay constraint when it fixes design rule violations.

Minimum delay constraints are set explicitly with the `set_min_delay` command or set implicitly due to hold time requirements.

The minimum delay to a pin or port must be greater than the target delay.

Design Compiler considers the minimum delay cost only if the `set_fix_hold` command is used.

If `fix_hold` is not specified on any clocks, the minimum delay cost is not considered during compilation. If `fix_hold` or `min_delay` is specified, the minimum delay cost is a secondary optimization cost.

`set_min_delay`

Defines a minimum delay for timing paths in the design.

`set_fix_hold`

Directs Design Compiler to fix hold violations at registers during compilation. You can override the default path delay for paths affected by `set_fix_hold`, by using the `set_false_path` or `set_multicycle_path` commands.

Hold time violations are fixed only if the `fix_hold` command is applied to related clocks.

Cost Calculation

The minimum delay cost for a design is different from the maximum delay cost. The minimum delay cost is not affected by path groups, and all violations contribute to the cost. Figure 7-6 shows the minimum delay cost equation.

Figure 7-6 Cost Calculation for Minimum Delay

$$\sum_{i=1}^{m} v_i$$

i = index
m = number of paths affected by set_min_delay or set_fix_hold
v = minimum delay violation max(0,required_path_delay - actual_path_delay)

The minimum delay cost function has the same delta for single pins or ports and multiple pins or ports.

```
Delta = min_delay - minimum delay(pin or port)
```

Maximum Area

Maximum area is an optimization constraint.

Maximum area represents the number of gates in the design, not the physical area the design occupies.

Usually the area requirements for the design are stated as the smallest design that meets the performance goal. Defining a maximum area directs Design Compiler to optimize the design for area after timing optimization is complete.

The `set_max_area` command specifies the maximum allowable area for the current design. Design Compiler computes the area of a design by adding the areas of each component on the lowest level of the design hierarchy (and the area of the nets).

Design Compiler ignores the following components when it calculates circuit area:

- Unknown components
- Components with unknown areas
- Technology-independent generic cells

The area of a cell (component) is technology-dependent and obtained from the technology library.

Cost Calculation

The maximum-area cost equation is

```
cost = max (0, current area - max_area)
```

Defining Maximum Area

The set_max_area command specifies an area target and places a max_area attribute on the current design.

Examples

```
dc_shell> set_max_area 0.0
dc_shell> set_max_area 14.0
```

Determining the smallest design can be helpful. The following script guides Design Compiler to optimize for area only. It constrains a design only for minimum area (when you do not care about timing). For the timing to make sense, you must apply clocking and input and output delay.

```
/* example script for smallest design */
remove_constraint -all
remove_clock -all
set_max_area 0
```

For more information, see the set_max_area man page.

Managing Constraint Priorities

During optimization, Design Compiler uses a cost vector to resolve any conflicts among competing constraint priorities. Table 7-2 shows the default order of priorities.

Table 7-2 Constraints Default Cost Vector

Priority (descending order)	Notes
connection classes	
multiple_port_net_cost	
min_capacitance	Design Rule Constraint
max_transition	Design Rule Constraint
max_fanout	Design Rule Constraint
max_capacitance	Design Rule Constraint
cell_degradation	Design Rule Constraint
max_delay	Optimization Constraint

Table 7-2 Constraints Default Cost Vector (Continued)

Priority (descending order)	Notes
min_delay	Optimization Constraint
power	Optimization Constraint
area	Optimization Constraint
cell count	

Table 7-2 shows that, by default, design rule constraints have priority over optimization constraints. However, you can reorder the priorities of the constraints listed in **bold** type, by using the `set_cost_priority` command. For example, the following are circumstances under which you might want to move the optimization constraint `max_delay` ahead of the maximum design rule constraints.

- In many technology libraries, the only significant design rule violations that cannot be fixed without hurting delay are overconstrained nets, such as input ports with large external loads or around logic marked `dont_touch`. Placing `max_delay` ahead of the design rule constraints in priority allows these design rule constraint violations to be fixed in a way that does not hurt delay. Design Compiler might, for example, resize the drivers in another module.
- In compilation of a small block of logic, such as an extracted critical region of a larger design, the possibility of overconstraints at the block boundaries is high. In this case, design rule fixing might better be postponed until the small block has been regrouped into the larger design.

The syntax is

```
set_cost_priority [-default] [-delay]  cost_list
```

`-default`

Directs Design Compiler to use its default priority, as shown in Table 7-2 on page 7-19.

`-delay`

Specifies that `max_delay` has higher priority than the maximum design rule constraints.

`cost_list`

Specifies the order of priority (listing the highest first) of the following costs: `max_delay`, `min_delay`, `max_transition`, `max_fanout`, `max_capacitance`, `cell_degradation`, and `max_design_rules`.

Note:
Use of the `cost_list` option requires a DC Ultra license.

Examples

To prioritize max_delay ahead of the maximum design rule constraints, enter

```
dc_shell> set_cost_priority -delay
```

To assign top priority to max_capacitance, max_delay, and max_fanout—in that order—enter

```
dc_shell> set_cost_priority {max_capacitance max_delay max_fanout}
```

Design Compiler assigns any costs you do not list to a lower priority than the costs you do list. This example does not list max_transition, cell_degradation, or min_delay, so Design Compiler assigns them priority following max_fanout.

If you specify set_cost_priority more than once on a design, Design Compiler uses the most recent setting.

Reporting Constraints

The report_constraint command reports the constraint values in the current design, enabling you to check design rules and optimization goals.

The constraint report lists the following for each constraint in the current design:

- Whether the constraint was met or violated
- By how much the constraint value was met or violated
- The design object that is the worst violator
- The maximum delay information, showing cost by path group. This includes violations of setup time on registers or ports with output delay as well as violations of the set_max_delay command.
- The minimum delay cost, which includes violations of hold time on registers or ports with output delay as well as violations of the set_min_delay command.

The constraint report summarizes the constraints in the order in which you set the priorities. Constraints not present in your design are not included in the report. To list more information in a constraint report, use the -verbose option of the report_constraint command. To list all constraint violators, use the -all_violators option of the report_constraint command.

For more information about the report_constraint command, see the man page.

Propagating Constraints in Hierarchical Designs

Hierarchical designs are composed of subdesigns. You can propagate constraints up or down the hierarchy in the following ways:

characterizing

Captures information about the environment of specific cell instances and assigns the information as attributes on the design to which the cells are linked.

modeling

Creates a characterized design as a library cell.

propagating constraints up the hierarchy

Propagates clocks, timing exceptions, and disabled timing arcs from lower level subdesigns to the current design.

Characterizing Subdesigns

When you compile subdesigns separately, boundary conditions such as the input drive strengths, input signal delays (arrival times), and output loads can be derived from the parent design and set on each subdesign. You can do this manually or automatically.

manually

Use the `set_drive`, `set_driving_cell`, `set_input_delay`, `set_output_delay`, and `set_load` commands.

automatically

Use the `characterize` command or the design budgeting tool.

Using the characterize Command

The `characterize` command places on a design the information and attributes that characterize its environment in the context of a specified instantiation in the top level design.

The primary purpose of `characterize` is to capture the timing environment of the subdesign. This occurs when you use `characterize` with no arguments or when you use its `-constraints`, `-connections`, or `-power` options.

The characterize command derives and asserts the following information and attributes on the design to which the instance is linked:

- Unless -no_timing is specified, characterize places on the subdesigns any timing characteristics previously set by the following commands:

create_clock	set_load
group_path	set_max_delay
read_timing	set_max_time_borrow
set_annotated_check	set_min_delay
set_annotated_delay	set_multicycle_path
set_auto_disable_drc_nets	set_operating_conditions
set_drive	set_output_delay
set_driving_cell	set_resistance
set_false_path	set_timing_ranges
set_ideal_net	set_wire_load_model
set_ideal_network	set_wire_load_mode
set_input_delay	set_wire_load_model_selection_group
	set_wire_load_min_block_size

- If you specify -constraint, characterize places on the subdesigns any area, power, connection class, and design rule constraints previously set by the following commands:

set_cell_degradation	set_max_fanout
set_connection_class	set_max_power
set_dont_touch_network	set_max_transition
set_fanout_load	set_min_capacitance
set_max_area	set_max_capacitance

- If you specify -connection, characterize places on the subdesigns the connection attributes set by the following commands (Connection class information is applied only when you use -constraint.):

set_equal	set_logic_zero
set_logic_dc	set_opposite
set_logic_one	set_unconnected

- If you specify -power, characterize places on the subdesigns the switching activity information, toggle rates, and static probability previously set, calculated, or saved by the following commands:

report_power	set_switching_activity

Removing Previous Annotations

In most cases, characterizing a design removes the effects of a previous characterization and replaces the relevant information. However, in the case of back-annotation (set_load, set_resistance, read_timing, set_annotated_delay, set_annotated_check), the characterize step removes the annotations and cannot overwrite existing annotations made on the subdesign. In this case, you must explicitly remove annotations from the subdesign (using reset_design) before you run the characterize command again.

Optimizing Bottom Up Versus Optimizing Top Down

During optimization, you can use characterize with set_dont_touch to maintain hierarchy. This is known as bottom-up optimization, which you can apply by using a golden instance or a uniquify approach (either manually with the uniquify command or automatically as part of the compile command). An alternative to bottom-up optimization is top-down optimization, also called hierarchical compile. During top-down optimization, the tool automatically performs characterization and optimization for subdesigns.

Deriving the Boundary Conditions

The characterize command automatically derives the boundary conditions of a subdesign based on its context in a parent design. It examines an instance's surroundings to obtain actual drive, load, and timing parameters and computes three types of boundary conditions:

timing conditions

Expected signal delays at input ports.

constraints

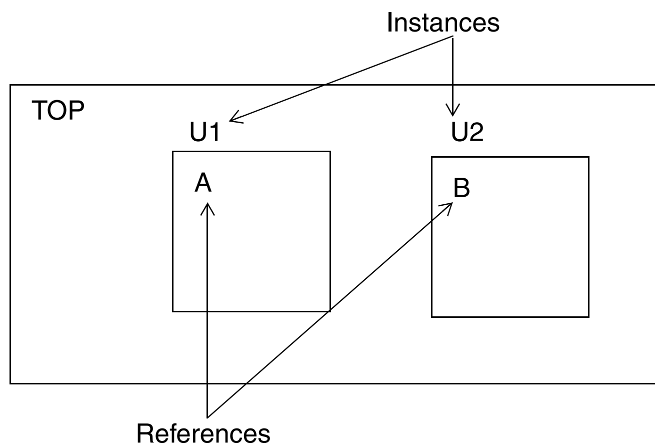
Inherited requirements from the parent design, such as maximum delay.

connection relations

Logical relationships between ports or between ports and power or ground, such as always logic 0, logical opposite of another port, or unconnected.

The `characterize` command summarizes the boundary conditions for one instance of a subdesign in one invocation. The result is applied to the reference. Figure 7-7 shows instances and references.

Figure 7-7 Instances and References



If a subdesign is used in more than one place, you must either characterize it manually or create a copy of the design for each instantiation and characterize each. See “Characterizing Multiple Instances” on page 7-33.

Limitations of the characterize Command

The `characterize` command provides many useful features, but do not always rely on this command to derive constraints for the subdesigns in a design hierarchy. Before you characterize a design, keep in mind the following limitations:

The `characterize` command

- Does not derive timing budgets. (It reflects the current state of the design.)
- Ignores `clock_skew` and `max_time_borrow` attributes placed on a hierarchical boundary (generally not an issue, because these attributes are usually placed on clocks and cells).

With no options, the `characterize` command replaces a subdesign’s port signal information (clocks, port drive, input and output delays, maximum and minimum path delays, and port load) with information derived from the parent design.

The characterize command recognizes when the top-level design has back-annotated information (load, resistance, or delay) and to move this data down to the subdesign in preparation for subsequent optimization.

Saving Attributes and Constraints

The characterize command captures the timing environment of the subdesign. Use the write_script command with characterize to save the attributes and constraints for the current design. By default, the write_script command writes the dc_shell commands to standard output. You can redirect the output of this command to a disk file by using the redirection operator (>).

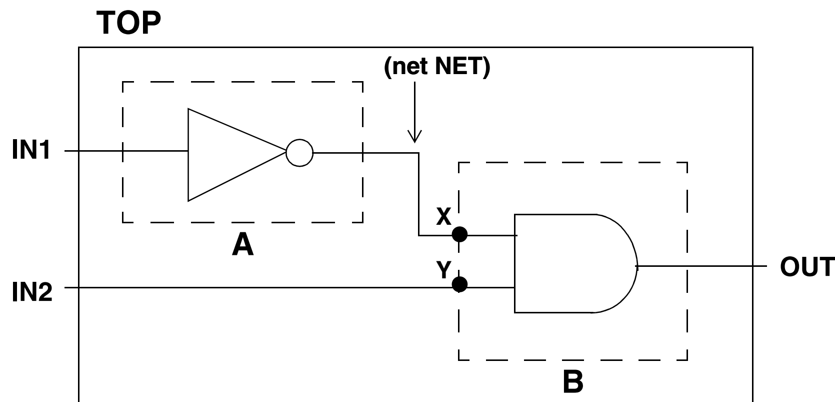
characterize Command Calculations

This section describes how the characterize command derives specific load and timing values for ports.

The characterize command uses values that allow the timing numbers on the output ports of a characterized design to be the same as if the design were flattened and then timed.

The "Load Calculations" and "Input Delay Calculations" sections that follow use the hierarchical design example in Figure 7-8 to describe the load and input delay calculations used by the characterize command.

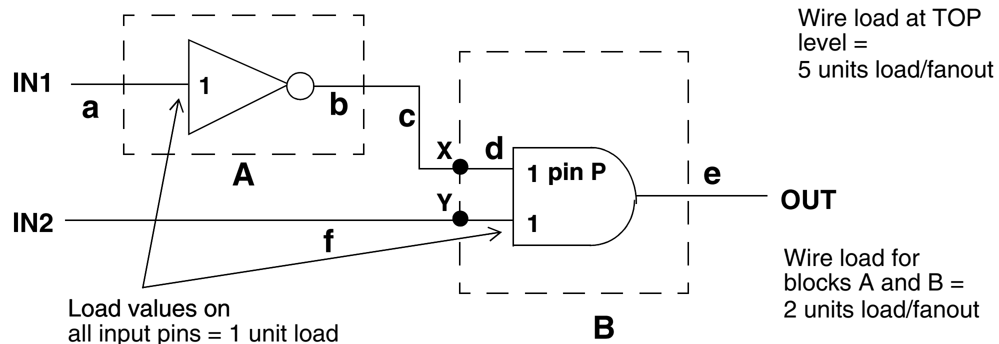
Figure 7-8 Hierarchical Design Example



Load Calculations

Figure 7-9 provides values for wire loads and input pin capacitances for the example shown in Figure 7-8.

Figure 7-9 Hierarchical Design With Annotated Loads



In Figure 7-9, a through f are wire segments used in the calculations. This example assumes a segmented wire loading model, which takes the interconnection net loads on the blocks into account and uses a linear function for the wire loads.

- The calculation for the outside load of pin P of hierarchical block B is

```
outside load = sum of the loads of all pins on the net
     loading P that are not in B
     plus the sum of the loads of all segments of net
     driving or loading P that are not in B
```

- The calculation for each segment's load is

```
segment load = number of fanouts * wire load
```

  - The calculation for the outside load on input IN1 to block A uses 0 driving pins, a fanout count of 1 for segment a, and the TOP wire load of 5 loads per fanout.

    The calculation is

```
load pins on driving net + load of segment a
     = 0 + (1 * 5)
     = 5
```

  - The calculation for the outside load on the output of block A is

```
load pins on net
          + load of segment c
          + load of segment d
     = 1 (for load pin P)
     + (1 * 5)
     + (1 * 2)
     = 8
```

For each segment in the calculation, the local wire load model is used to calculate the load. That is, the calculation for block A's output pin uses TOP's wire load of 5 loads per fanout for segment c and block B's wire load of 2 loads per fanout for segment d.

- The calculation for the outside load on the output of block B is

```
load pins on net + load of segment e
      = 0 + (1 * 5)
      = 5
```

- The calculation for the outside load on the input pin X of block B is

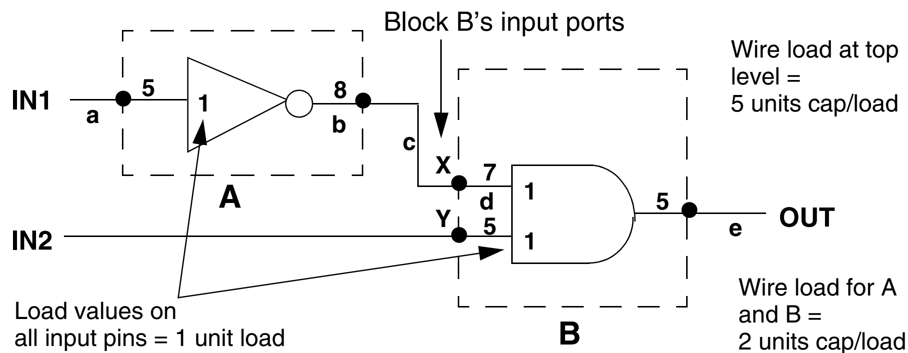
```
load pins on net
            + load of segment b
            + load of segment c
            = 0 + (1 * 2) + (1 * 5)
            = 7
```

- The calculation for the outside load on the input pin Y is

```
pin loads on driving net + load of segment f
      = 0 + 1 * 5
      = 5
```

Figure 7-10 shows the modified version of Figure 7-8 with the outside loads annotated.

Figure 7-10 Loads After Characterization



Input Delay Calculations

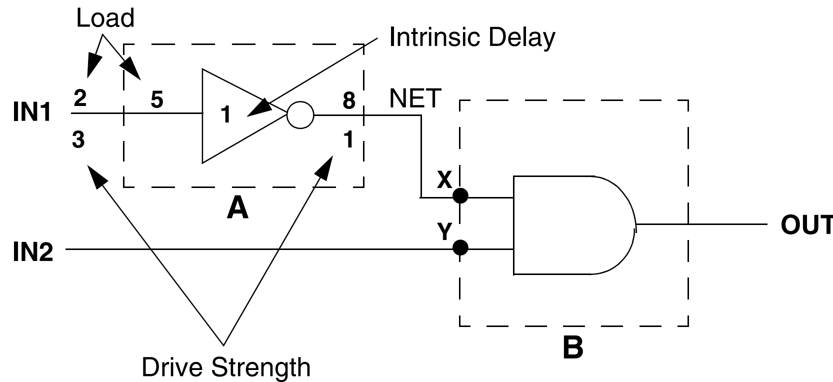
Because characterizing provides accurate details of outside loads, the path delays of input signals reflect only the delay through the intrinsic delay of the last gate driving the port. The path delays of input signals do not include the gate's load delay or the connect delay on the net.

For example, the characterized input delay on the input pins of block B is calculated from the delay to the pin that drives the port being characterized, without the gate's load delay or the connect delay on the net.

The timing calculations for characterizing block B follow Figure 7-11.

Figure 7-11 shows the default drive strengths and intrinsic delays of block A and signal IN1.

Figure 7-11 Design With Annotations for Timing Calculations



The delay calculation for input pin X is

```
drive strength at IN1 * (wire load + pin load) + intrinsic delay of A’s
cell     = 3 * (5 + 2 + 1) + 1 = 25
```

Characterizing Subdesign Port Signal Interfaces

The `characterize` command with no options replaces a subdesign’s port signal information (clocks, port drive, input and output delays, maximum and minimum path delays, and port load) with information derived from the parent design. The subdesign also inherits operating conditions and timing ranges from the top-level design.

You can manually set port signal information by using the following commands:

```
create_clock
set_clock_latency
set_clock_uncertainty
set_drive
set_driving_cell
set_input_delay
set_load
set_max_delay
set_min_delay
set_output_delay
set_propagated_clock
```

The `characterize` command sets the wire loading model selection group and model for subdesigns. The subdesigns inherit the top-level wire loading mode. The wire loading model for subdesigns is determined as follows:

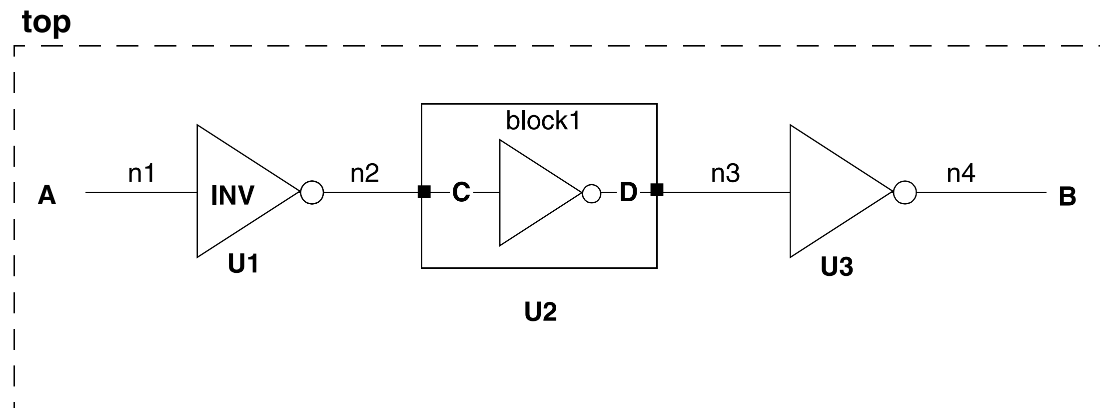
- If the top-level mode is top, the subdesigns inherit the top-level wire loading model. Unless an instance specific model or selection group has be specified. Instance specific model and selection group settings take precedence over design level settings.
- If the top-level mode is enclosed or segmented, the wire loading model is based on the following:
  - If no wire loading model is defined for the lower block and the wire load cannot be determine by the area, the wire loading model of the top-level design is used.
  - If no wire loading model is defined for the lower block but the wire load can be determine by the area, the wire loading model is reselected at compile time, based on the cell area.

Combinational Design Example

Figure 7-12 shows subdesign block in the combinational design top. Set the port interface attributes either manually or automatically.

- Script 1 uses characterize to set the attributes automatically.
- Script 2 sets the attributes manually.

Figure 7-12 Characterizing Drive, Timing, and Load Values—Combinational Design




```
current_design top
set_input_delay 0 A
set_max_delay 10 -to B
```

Script 1

```
current_design top
characterize U2
```

Script 2

```
current_design block1
set_driving_cell -lib_cell INV {C}
set_input_delay 3.3 C
set_load    1.3 D
set_max_delay 9.2 -to D
current_design top
```

In Script 2,

Line 2 captures driving cell information.

Line 3 sets the arrival time of net n2 as 3.3.

Line 4 sets the load of U3 as 1.3.

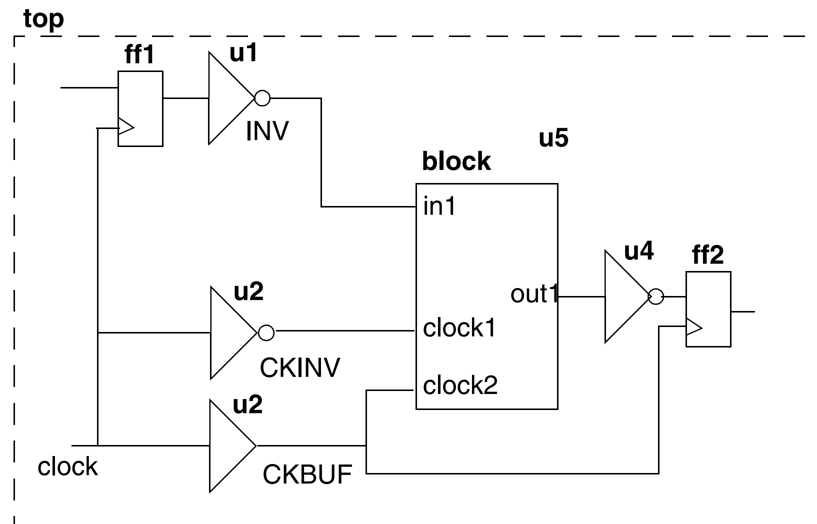
Line 5 sets the inherited `set_max_delay`, which is 10 – .8 (.4 for each inverter).

Sequential Design Example

Figure 7-13 shows subdesign block in the sequential design top. Set the port interface information either manually or automatically.

- Script 1 sets the information manually.
  - The input delay (from ff1/CP to u5/in1) is 1.8
  - The output delay (through u4 plus the setup time of ff2) is 1.2
  - The outside load on the out1 net is 0.85
- Script 2 uses `characterize` to set the information automatically.

Figure 7-13 Characterizing Sequential Design Drive, Timing, and Load Values



Script 1

```
current_design top
create_clock -period 10 -waveform {0 5} clock
current_design block
create_clock -name clock -period 10 -waveform {0 5} clock1
create_clock -name clock_bar -period 10 \
   -waveform {5 10} clock2
set_input_delay -clock clock 1.8 in1
set_output_delay -clock clock 1.2 out1
set_driving_cell -lib_cell INV -input_transition_rise 1 in1
set_driving_cell -lib_cell CKINV clock1
set_driving_cell -lib_cell CKBUF clock2
set_load 0.85 out1
current_design top
```

Script 2

```
current_design top
create_clock -period 10 -waveform {0 5} clock
characterize u5
```

Characterizing Subdesign Constraints

The `characterize -constraints` command uses values derived from the parent design to replace the `max_area`, `max_fanout`, `fanout_load`, `max_capacitance`, and `max_transition` attributes of a subdesign.

Characterizing Subdesign Logical Port Connections

The `characterize -connections` command uses connection attributes derived from the parent design to replace the port connection attributes of a subdesign.

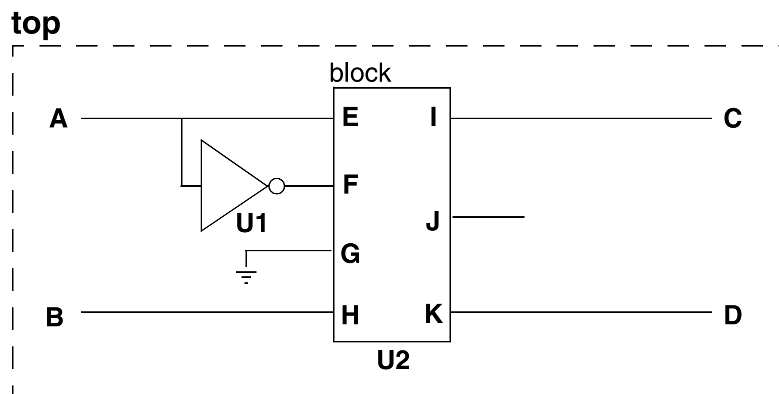
The connection attributes are those set by the commands `set_equal`, `set_opposite`, `set_logic_one`, `set_logic_zero`, and `set_unconnected`.

Example

Figure 7-14 shows subdesign block in design top. Set logical port connections either manually or automatically.

- Script 1 sets the attributes manually.
- Script 2 uses `characterize -no_timing -connections` to set the attributes automatically. The `-no_timing` option inhibits computation of port timing information.

Figure 7-14 Characterizing Port Connection Attributes



Script 1

```
current_design block
set_opposite    { E F }
set_logic_zero  { G }
set_unconnected { J }
```

Script 2

```
current_design top
characterize U2 -no_timing -connections
```

Characterizing Multiple Instances

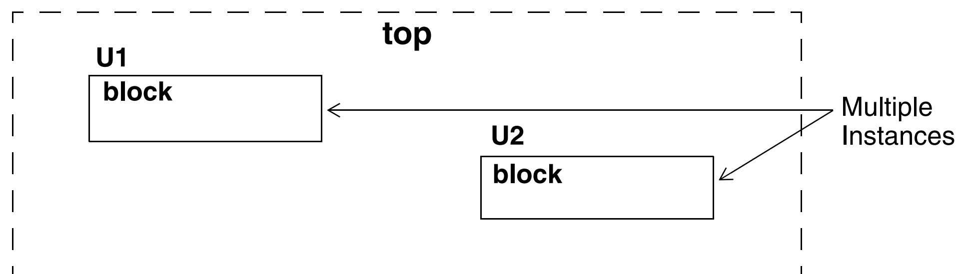
The `characterize` command summarizes the boundary conditions for one instance of a subdesign in each invocation. If a subdesign is used more than once, use the `uniquify` command to make each instance distinctive before using `characterize` for each instance.

The `uniquify` command creates copies of subdesigns that are referenced more than once. It then renames the copies and updates the corresponding cell references. For information about the `uniquify` command, see the man page for the command.

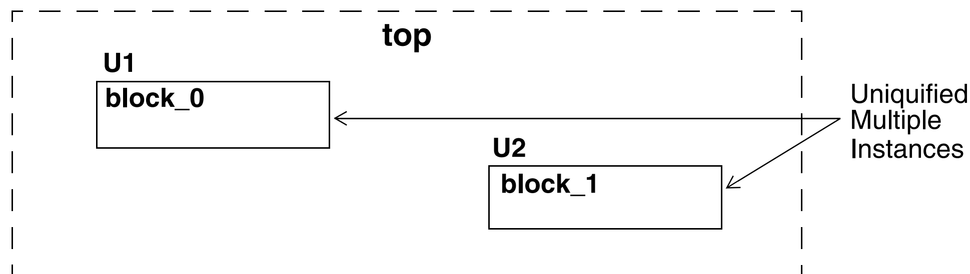
Example

Figure 7-15 shows how to use `uniquify` and `characterize` for the subdesign block, which is referenced in cells U1 and U2.

Figure 7-15 Characterizing a Subdesign Referenced Multiple Times




```
dc_shell> current_design top
dc_shell> uniquify -reference block
```




```
dc_shell> characterize U1
dc_shell> characterize U2
```

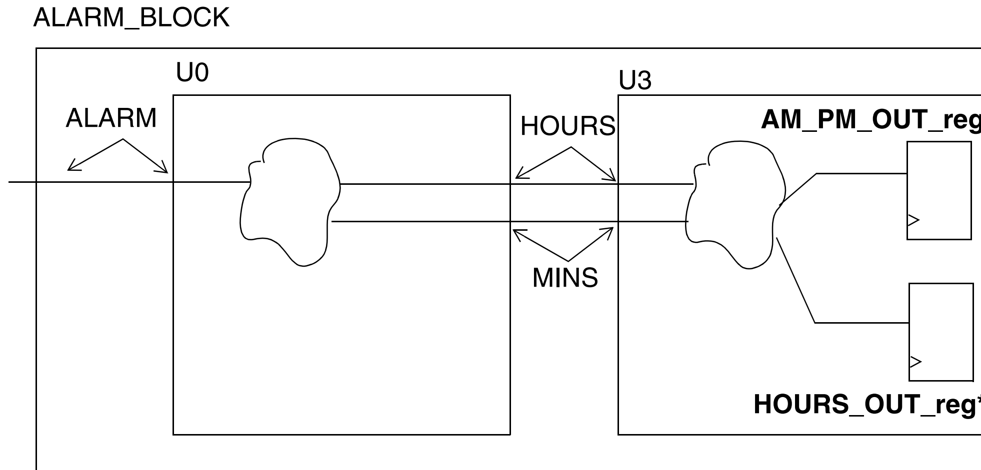
Characterizing Designs With Timing Exceptions

When paths crossing design hierarchies contain different timing exceptions, `characterize` creates timing constraints with virtual clocks to capture this information.

Possible timing exceptions include `set_multicycle_path`, `set_false_path`, `set_max_delay`, and `set_min_delay`. The virtual clock scheme can also handle multiple clocks.

Figure 7-16 shows a sample design in which the path from the ALARM input port to HOURS_OUT_reg* in U3 is constrained as a two-cycle path. Example 7-1 shows the a portion of the uncompressed `characterize -verbose` result of U0 block.

Figure 7-16 characterize -verbose Result of U0 Block



Example 7-1 characterize -verbose Output

```
create_clock -period 10 -waveform {0 5} [get_ports {CLK}]
create_clock -name "CLK_virtual1" -period 10 -waveform {0 5}
create_clock -name "CLK_virtual2" -period 10 -waveform {0 5}
create_clock -name "CLK_virtual3" -period 10 -waveform {0 5}
create_clock -name "CLK_virtual4" -period 10 -waveform {0 5}
create_clock -name "CLK_virtual5" -period 10 -waveform {0 5}
set_input_delay 2 -clock "CLK" [get_ports {MINUTES_BUTTON}]
set_input_delay 2 -clock "CLK" [get_ports {HOURS_BUTTON}]
set_input_delay 2 -clock "CLK_virtual1" [get_ports {ALARM_BUTTON}]
set_output_delay 7.62796 -max -rise -clock "CLK" [get_ports {MINS}]
set_output_delay 6.93955 -max -fall -clock "CLK" [get_ports {MINS}]
set_output_delay 2.20324 -min -rise -clock "CLK" [get_ports {MINS}]
set_output_delay 2.40013 -min -fall -clock "CLK" [get_ports {MINS}]
set_output_delay 8.05575 -add_delay -max -rise -clock "CLK_virtual2" \
   [get_ports {MINS}]
set_output_delay 6.83933 -add_delay -max -fall -clock "CLK_virtual2" \
   [get_ports {MINS}]
set_output_delay 2.89679 -add_delay -min -rise -clock "CLK_virtual2" \
   [get_ports {MINS}]
set_output_delay 2.80452 -add_delay -min -fall -clock "CLK_virtual2" \
   [get_ports {MINS}]
...
set_output_delay 10.232 -add_delay -max -rise -clock "CLK_virtual5" \
   [get_ports {MINS}]
set_output_delay 9.4791 -add_delay -max -fall -clock "CLK_virtual5" \
   [get_ports {MINS}]
set_output_delay 3.90222 -add_delay -min -rise -clock "CLK_virtual5" \
   [get_ports {MINS}]
set_output_delay 3.57818 -add_delay -min -fall -clock "CLK_virtual5" \
   [get_ports {MINS}]
...
set_multicycle_path 2 -from [get_clocks {CLK_virtual1}] -to \
   [get_clocks {CLK_virtual2}]
set_multicycle_path 2 -from [get_clocks {CLK_virtual1}] -to \
   [get_clocks {CLK_virtual3}]
set_multicycle_path 2 -from [get_clocks {CLK_virtual1}] -to \
   [get_clocks {CLK_virtual4}]
set_multicycle_path 2 -from [get_clocks {CLK_virtual1}] -to \
   [get_clocks {CLK_virtual5}]
```

Propagating Constraints up the Hierarchy

If you have hierarchical designs and compile the subdesigns, then move up to the higher-level blocks (bottom-up compilation), you can propagate clocks, timing exceptions, and disabled timing arcs from lower-level .ddc files to the current design, using the `propagate_constraints` command.

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

Methodology for Propagating Constraints Upward

Use the `propagate_constraints` command to propagate constraints from lower levels of hierarchy to the current design. If you do not use the command, you can propagate constraints from a higher-level design to the `current_instance`, but you cannot propagate constraints set on a lower-level block to the higher-level blocks in which it is instantiated.

Note:
Using the `propagate_constraints` command might cause memory usage to increase.

Example 7-2 shows a methodology in which constraints are propagated upwards. Assume that A is the top-level and B is the lower-level design.

Example 7-2 Propagating Constraints Upward

```
current_design B
source constraints.tcl
compile
current_design A
propagate_constraints -design B
report_timing_requirements
compile
report_timing
```

To generate a report of all the constraints that were propagated up, use the `-verbose` and `-dont_apply` options and redirect the output to a file:

```
dc_shell> propagate_constraints -design name \
           -verbose -dont_apply -output report.cons
```

Use the `-write -format ddc` command to save the .ddc file with the propagated constraints so there is no need to go through the propagation again when restarting a new dc_shell session.

Handling of Conflicts Between Designs

The following shows what happens if there are conflicts between the lower-level and top-level designs.

Clock name conflict

The lower-level clock has the same name as the clock of the current design (or another block).

The clock is not propagated. A warning is issued.

Clock source conflict

A clock source of a lower-level block is already defined as a clock source of a higher-level block.

The lower-level clock is not propagated. A warning is issued.

Exceptions from or to an unpropagated clock

This can be either a virtual clock, or a clock that was not propagated from that block due to a conflict.

Exceptions

A lower-level exception overrides a higher-level exception that is defined on the exact same path.

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. You use the `compile_ultra` command or the `compile` command to compile a design. Design Compiler provides options that enable you to customize and control optimization. Several of the many factors affecting the optimization outcome are discussed in this chapter. For detailed information, see the *Design Compiler Optimization Reference Manual*.

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

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.

- 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.

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 options of the `compile_ultra` command or the `compile` command 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_ultra` command or `compile` command. However, if you use this option, your synthesized design might violate design rules.

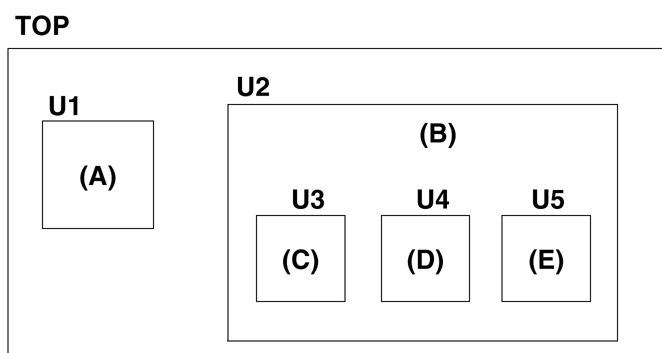
Selecting and Using a Compile Strategy

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 8-1, are defined by the script of Example 8-1. These specifications apply to TOP and all its subdesigns.

Table 8-1 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 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 -lib_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 the “Using Interface Logic Models” in Chapter 9.

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 the Chapter 9, “Using Interface Logic Models.

1. Read in the entire design.
2. 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 5-40. 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.

3. Compile the design.

A top-down compile script for the TOP design is shown in Example 8-2.The script contains comments that identify each of the steps. The constraints are applied by including the constraint file (defaults.con) shown in Example 8-1 on page 8-5.

Example 8-2 Top-Down Compile Script

```
/* 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 .ddc 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 .ddc 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).
- The subdesign constraints are not preserved after you perform a top-level compile. To ensure that you are using the correct constraints, always reapply the subdesign constraints before compiling or analyzing a subdesign.
- By default, compile modifies the original copy in the current design. This could be a problem when the same design is referenced from multiple modules, which are compiled separately in sequence. For example, the first compile could change the interface or the functionality of the design by boundary optimizaiton. When this design is referenced from another module in the subsequent compile, the modified design is uniquified and used.

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

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

A bottom-up compile script for the TOP design is shown in Example 8-3 on page 8-9. The script contains comments that identify each of the steps in the bottom-up compile strategy. In the script it is assumed that block constraint files exist for each of the subblocks (subdesigns) in design TOP. The compile script also uses the default constraint file (defaults.con) shown in Example 8-1 on page 8-5.

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 8-3 Bottom-Up Compile Script

```
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 -format ddc -output first_pass.ddc

# 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.

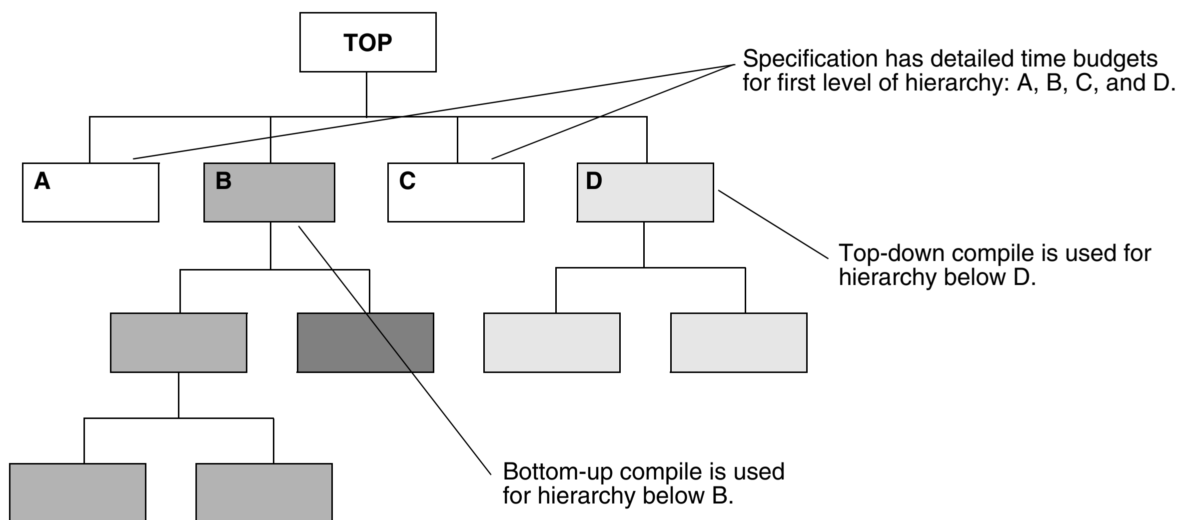
Mixed Compile Strategy

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.

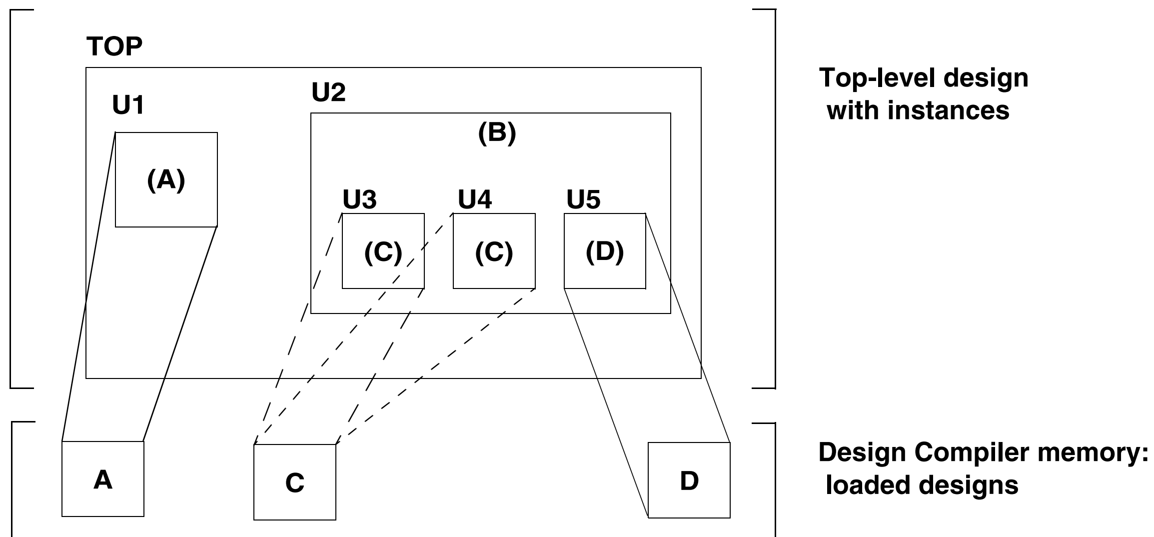
Figure 8-2 Mixing Compilation Strategies



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



Use the `check_design -multiple_design` command to report information messages related to multiply-instantiated designs. The command lists all multiply instantiated designs along with instance names and associated attributes (`dont_touch`, `black_box`, and `ungroup`). 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 compile the design. 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. The process removes the original design from memory after it creates the new, unique designs. The original design and any collections that contain it or its objects are no longer accessible.

In earlier releases, you had manually to run the `uniquify` command to create a uniquely named copy of the design for each instance. However, beginning with version V-2004.06, the tool automatically uniquifies designs as part of the compile process. The uniquification process can resolve multiple references throughout the hierarchy the current design (except those having a `dont_touch` attribute). After this process finishes, 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 of the `uniquify` command, or you can specify specific cells by using the `-cell` option. Design Compiler makes unique copies for cells specified with the `-reference` or the `-cells` option, even if they have a `dont_touch` attribute.

The `uniquify` command accepts instance objects—that is, cells at a lower level of hierarchy. When you use the `-cell` option with an instance object, the complete path to the instance is uniquified. For example, the following command uniquifies both instances mid1 and mid1/bot1 (assuming that mid1 is not unique):

```
dc_shell> uniquify -cell mid1/bot1
```

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`

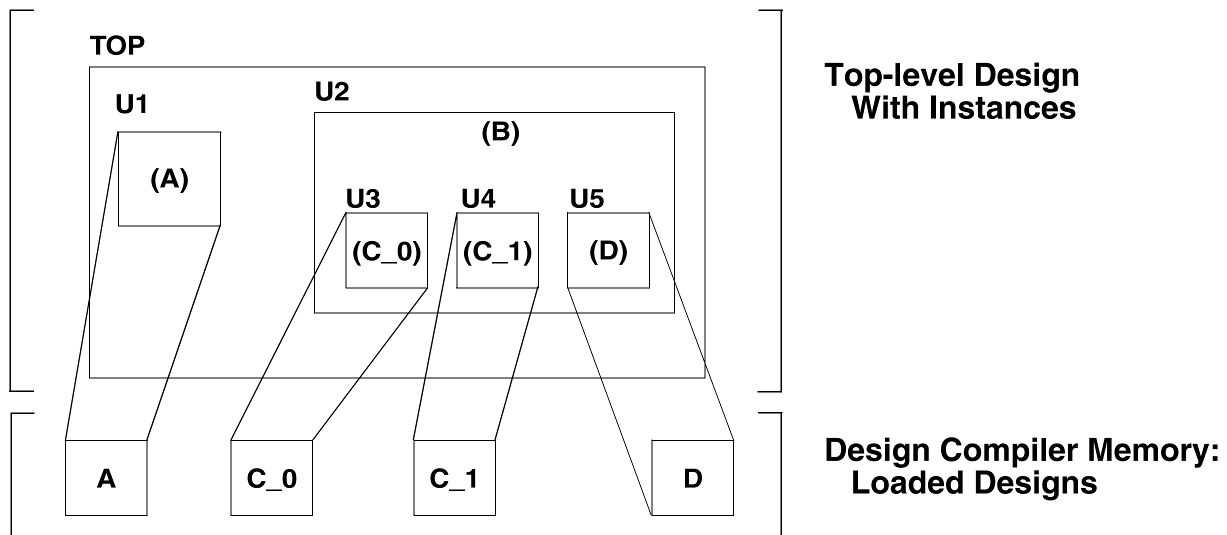
The smallest integer value that forms a unique subdesign name.

The following command sequence resolves the multiple instances of design C in design TOP shown in Figure 8-3 on page 8-12; 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 8-17.

To use the compile-once-don't-touch method to resolve multiple instances, follow these steps:

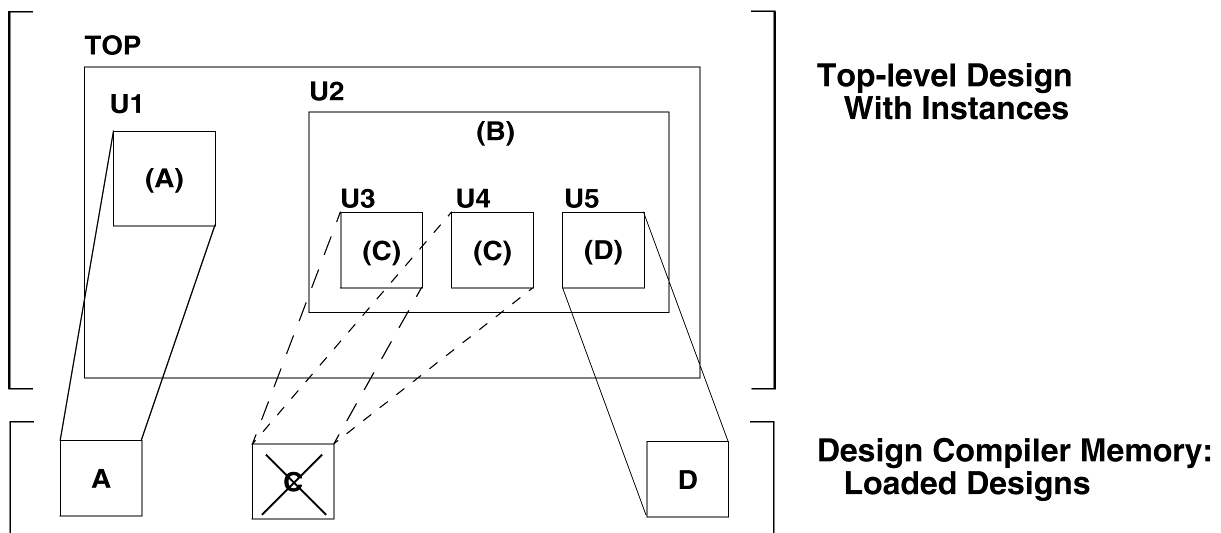
1. Characterize the subdesign's instance that has the worst-case environment.
2. Compile the referenced subdesign.
3. Use the `set_dont_touch` command to set the `dont_touch` attribute on all instances that reference the compiled subdesign.
4. Compile the entire design.

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

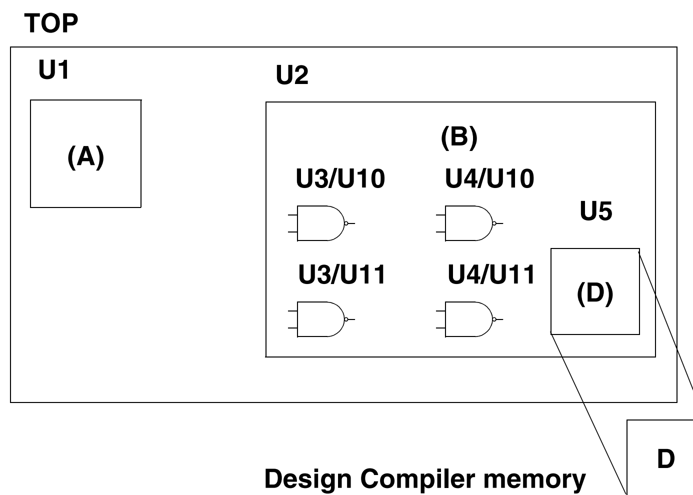
The ungroup method has the same effect as the uniquify method (it makes unique copies of the design), but in addition, it removes levels of hierarchy. This method uses the `ungroup` command to produce a flattened netlist. For details about the `ungroup` command, See "Removing Levels of Hierarchy" on page 5-25.

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 the Chapter 9, “Using Interface Logic Models.

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
1
dc_shell> report_design

****************************************
Report : design
Design : SUB_A
Version: Y-2006.06
Date   : Mon May 1 10:56:49 2006
****************************************

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

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_ultra` command or `compile` command.

You can disable evaluation of the optimization cost function by using the `-only_design_rule` option when running the `compile_ultra command or compile` command.

For more information on the compile cost function, see the *Design Compiler Optimization Reference Manual*.

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. 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 High-Performance Designs
- Optimizing for Maximum Performance
- Optimizing for Minimum Area
- Optimizing Data Paths

Optimizing High-Performance Designs

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

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 need a DC Ultra license and a DesignWare Foundation license.

For more information on this command, see the man page and the *Design Compiler Optimization Reference Manual*.

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
- 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

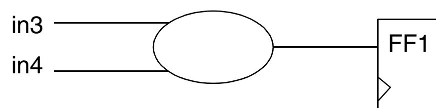
Controlling the Optimization of Your Design

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.

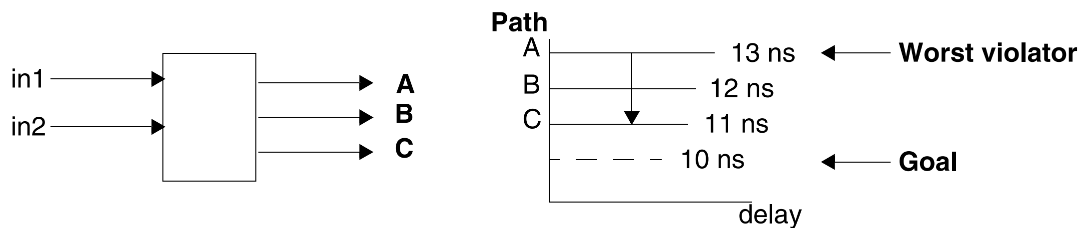
Specifying a critical range can increase runtime. To limit the runtime increase, use critical range only during the final implementation phase of the design, and use a reasonable critical range value. A guideline for the maximum critical range value is 10 percent of the clock period.

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. For example,

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

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.

Use the following script to create a path group for each endpoint.

```
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
}
```

Fixing Heavily Loaded Nets

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

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

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

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

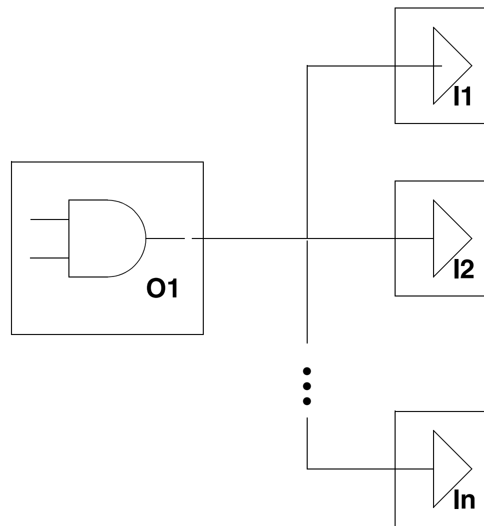
- If the large loads reside across the hierarchy from several modules, apply design rules to fix the problem. For example,

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

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



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, use the `compile_ultra` command or the -auto_ungroup delay option of the `compile` command.

For more information on auto-ungrouping, see the *Design Compiler Optimization Reference Manual*.

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. 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.

Incremental compile supports adaptive retiming, that is, `compile_ultra -inc -retime`.

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
- 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
```

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:

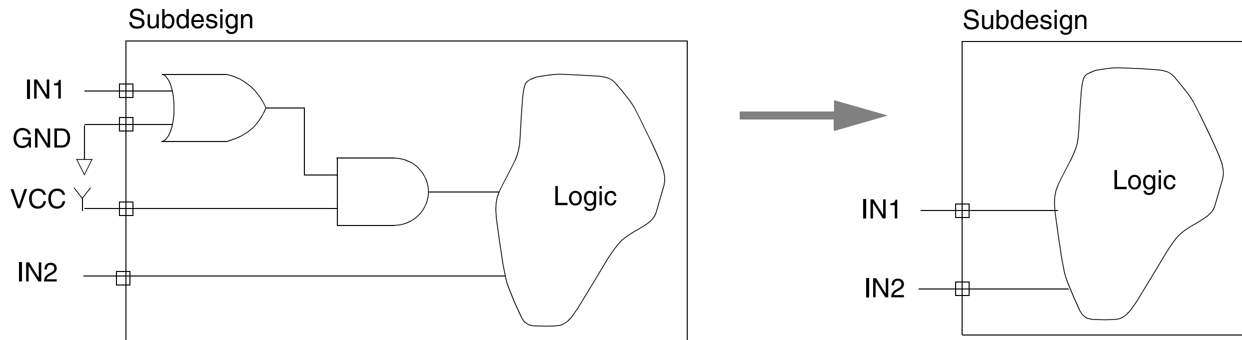
```
dc_shell> compile -boundary_optimization
```

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 8-10 shows this relationship.

Figure 8-10 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 5-25.

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 an improved datapath generator 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. If necessary, use the set synthetic_library dw_foundation.sldb command. These enhancements require a DesignWare license.

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 when you use `compile_ultra`.

To disable DC Ultra datapath optimization, set `hlo_disable_datapath_optimization` to true. (The default is false.)

For more information on datapath synthesis, see the *Design Compiler Optimization Reference Manual*.

9

Using Interface Logic Models

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. Interface logic models enhance capacity and reduce runtime for top-level optimization.

This chapter contains the following sections:

- Overview of Interface Logic Models (ILMs)
- General Guidelines for Creating ILMs
- Controlling the Logic Included in ILMs
- Methodology for Creating and Saving ILMs
- Instantiating and Using ILMs in the Top-Level Design
- Using the create_ilm Command Options
- Using ILMs with Multicorner-Multimode Designs
- Reporting Information About ILMs
- Summary of Commands for ILMs

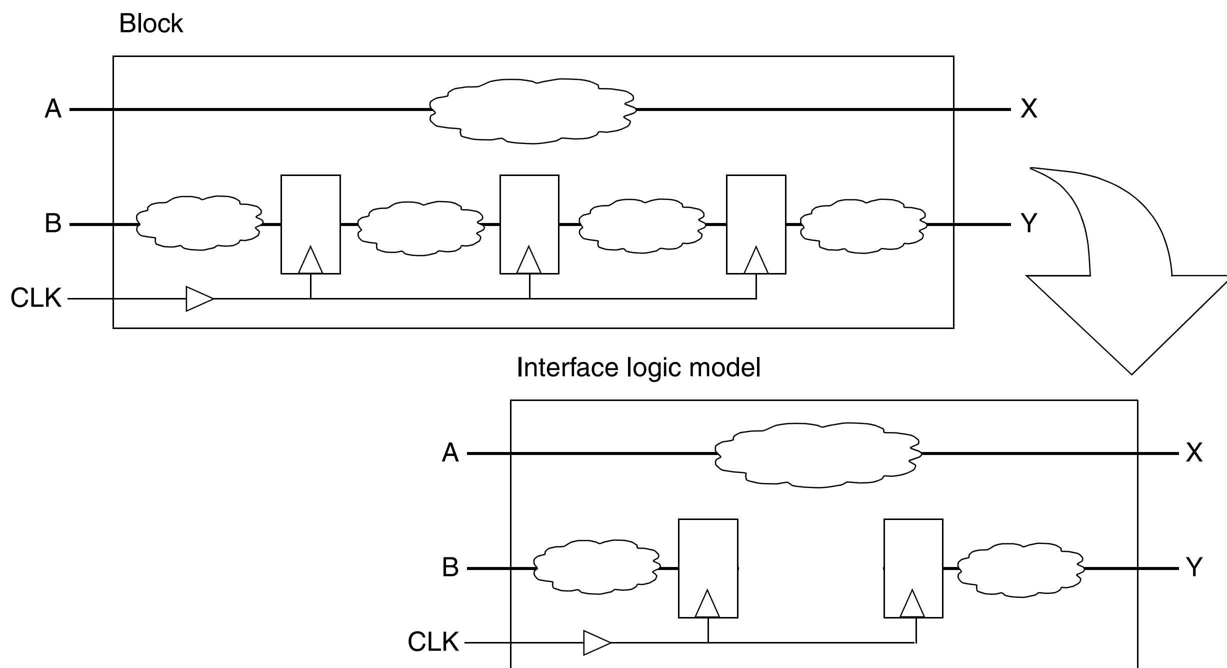
Overview of Interface Logic Models (ILMs)

Interface logic models (ILMs) are used in Design Compiler to reduce the number of design objects and memory required when performing top-level optimization on large designs. In an interface logic model, the *gate-level* netlist for a block is modeled by another *gate-level* netlist that contains the interface logic of a block and possibly logic associated with the interface logic. Logic not associated with the interface is removed.

ILMs are created by using the `create_ilm` command. Various options are available that let you control specific properties of the model. Also, for top-level design optimization, the properties of the ILMs are communicated to the top level by using the `propagate_ilm` command. These are the principal commands associated with using ILMs.

Figure 9-1 shows a block and its interface logic model. The logic between the input port and the first register is preserved, as is the logic between the last register and the output port. Clock connections to the preserved registers are kept as well. Logic associated with pure combinational input-port-to-output-port timing paths (A to X) is also preserved. The remaining logic is discarded. The timing exceptions specified on core logic (register-to-register) of a design are lost when that design becomes an ILM, because the core logic is not part of the model.

Figure 9-1 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 details of hierarchical test synthesis, see the DFT Compiler documentation.

Important:
Synthesis interface logic models support only SDF and `set_load` information for back-annotation. In topographical mode, block-level cell placement information is also back-annotated.

Benefits of Using ILMs

In addition to improved capacity and reduced compile times for very large designs when interface logic models are used, ILMs provide the following benefits:

- ILMs preserve interface logic without modifying it. In other words, this model does not abstract the design but instead discards only what is not required for modeling boundary timing. Any subblock in the hierarchy that affects the boundary timing is retained.
- ILMs provide, therefore, highly accurate timing representations, because they preserve the boundary interface logic.
- Differences in the timing characteristics of an interface logic model, when compared with the original netlist, are easy to debug because the interface logic is maintained. Clock and data paths are preserved without modification, and cell and net names are identical, enabling you to isolate the cause for discrepancies.
- The runtime for generating interface logic models is fast because identifying the interface logic for a design is a structural operation performed by analysis of circuit topology.
- A model is context independent because it is an abstraction of the original netlist and contains instantiations of library cells that comprise the interface logic.
- An interface logic model can be used for an instance nested anywhere in the design hierarchy, including within another interface logic model. For nested ILMs, you can control whether the tool includes entire lower-level ILM logic or only the lower-level ILM logic that is involved in the timing paths of the upper-level ILM.
- 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.
- ILMs can be saved as .ddc files without loss of attributes related to the interface logic.

General Guidelines for Creating ILMs

When creating ILMs, observe the following guidelines:

- 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 will have the greatest reduction in size (original netlist compared with its ILM).

- User-controlled handling of side-load cells (capacitance) can affect the timing of an interface path even when side-load cells are not in the interface logic.

- During chip-level optimizations, optimization is not performed on the logic or cells within an ILM.

- 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.

- Propagated clocks retain side loads if the appropriate value of the `create_ilm -include_side_load` option is set.

- If you want to create a compact ILM, you should remove the `dont_touch` attribute from any block-level subdesign that has this attribute set on it. 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 message.

- Use the `report_annotated_delay` command to check if the necessary nets and cells are annotated for a postroute SDF flow.

- 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.

- If the current design contains nested ILMs, 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.
- Designs containing ILMs can be budgeted. For more information about budgeting with ILMs, see the section "Budgeting With Interface Logic Models" in the *Budgeting for Synthesis User Guide*.

Controlling the Logic Included in ILMs

The following sections discuss controlling the logic included in ILMs:

- Basic ILM Logic Content
- Controlling the Fanins and Fanouts of Chip-Level Networks
- Controlling the Number of Latch Levels
- Controlling Side-Load Cells

Basic ILM Logic Content

The basic interface logic model contains the following circuitry:

- 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 can be included in the model by using the `-keep_macros` option of the `create_ilm` command.

  This capability is useful when an ILM is taken into a floorplanning 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.

- The boundary cells of ignored ports that you include by using the -keep_boundary_cells option of the create_ilm command. (Ignored ports are specified by using the -ignore_ports or -no_auto_ignore option.)
- Clock trees that drive interface registers, including registers in the clock tree.

  To include the entire clock tree, use the -keep_full_clock_tree option of the create_ilm command. Note that using this option can increase the memory requirement of the ILM significantly.
- 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 9-2 on page 9-7 and Figure 9-3 on page 9-8.)
- 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. 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 (for example, by commands such as set_false_path, set_disable_timing) 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.

Note:
The cells, nets, and pins that are identified as part of an ILM for the current design are marked with the is_interface_logic attribute.

Controlling the Fanins and Fanouts of Chip-Level Networks

Usually you do not want the fanins and fanouts of chip-level networks, such as scan enable and set/reset networks, to be part of the ILM. 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 -no_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 ILM, 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, 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 9-2 shows an ILM default model.

Figure 9-2 ILM Default Model

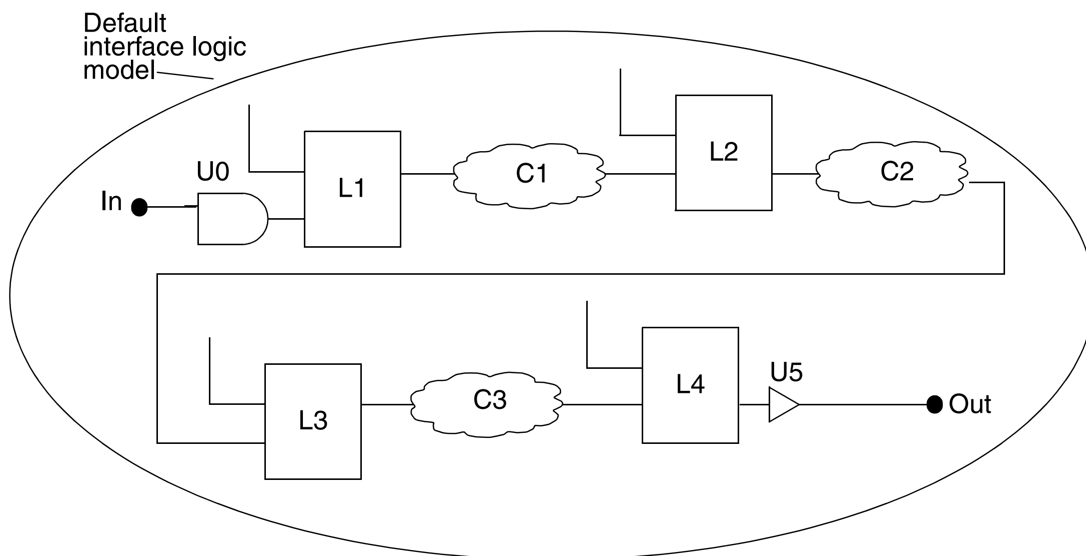
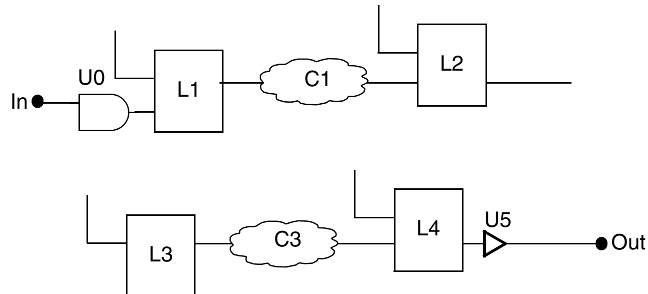


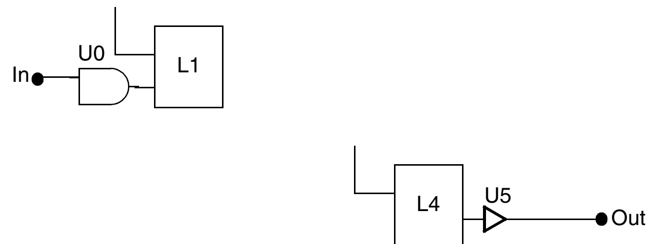
Figure 9-3 shows the results of specifying `-latch_level 1` and `-latch_level 0`.

Figure 9-3 Results of Specifying -latch_level 1 and -latch_level 0

Interface logic model
-latch_ level = 1



Interface logic model
-latch_level = 0

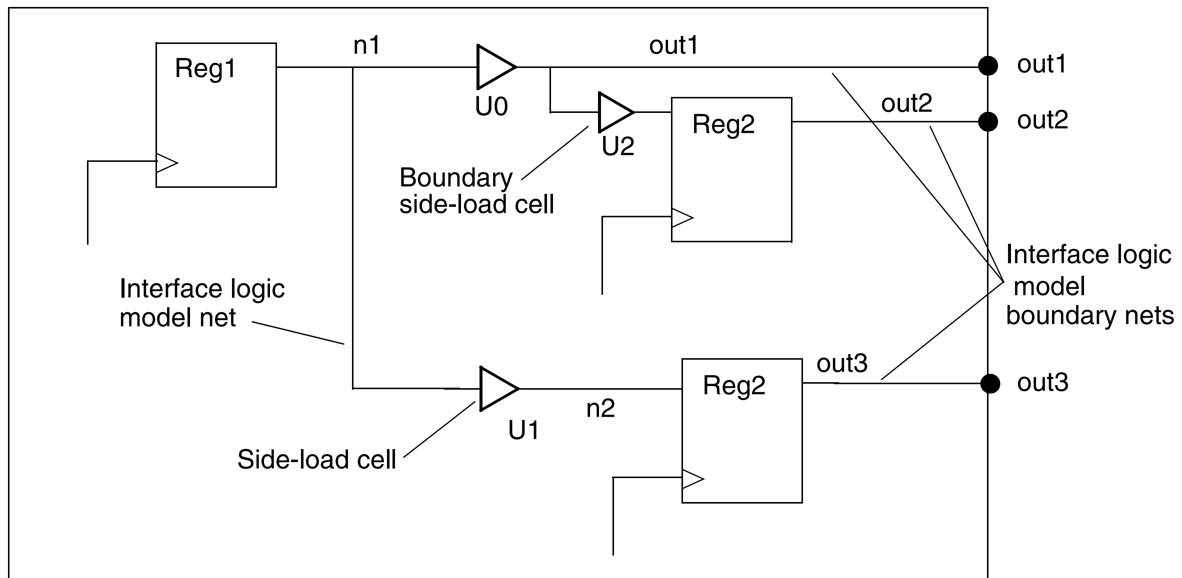


Controlling Side-Load Cells

By default, only side-load cells that are boundary cells are included in an ILM. 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 9-4 shows a side-load cell and a boundary side-load cell.

Figure 9-4 Side-Load Cell and Boundary Side-Load Cell



In Figure 9-4, 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 ILM net out1. Net out1 is considered a boundary net because it is directly connected to a port (out1). To include all side-load cells in the ILM, use the `create_ilm -include_side_load all` command. Using this command increases the model size but can improve model timing accuracy.

Methodology for Creating and Saving ILMs

You use the `create_ilm` command to generate an ILM. A number of command options are available that allow you to control the details of how the interface logic is modeled. The `create_ilm` command does the following:

- 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.
- Automatically marks the ILM with the `is_interface_model` attribute. Prior to optimizing the top-level design, you use the `propagate_ilm` command to propagate information about the ILM blocks to the top level.Exceptions related to the interface are saved in the model but are not automatically visible to the top-level design.

Note:
The propagate_ilm command is not supported in topographical mode; information about ILM blocks is automatically propagated in this topographical mode.

To create ILMs, use the following steps:

1. Determine the block(s) you want to model as ILMs.
2. Read each mapped block design that you want to model and link each design.
3. Identify all clocks by using the appropriate commands, for example, create_clock or create_generated_clock.
4. Use the write_interface_timing command to create a timing report for the original netlist. This timing report will be compared to the timing report for the ILM to help verify consistency between the ILM model and the original netlist.
5. (Optional) If you are using multi-scenarios, see Example 9-1 for the flow to use to verify consistency between the original netlist and the ILM.
6. Create the ILM for each block you chose. Use the create_ilm command with appropriate options as needed.

   When you execute 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 issues 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.

   After command execution, the design in memory is the ILM for the block. It has the same name as the original design and becomes 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, use the create_ilm -identify_only and create_ilm -extract_only commands. For details, see “Controlling the Logic Included in ILMs” on page 9-5.

7. (Optional) Use the get_ilm_objects command to determine the objects that have been identified as interface logic objects. You can repeat this step, changing option settings as needed, until you obtain the desired collection of interface objects. You can then use the create_ilm -extract_only command to extract the currently identified interface logic objects and build the ILM with these objects. The create_ilm -extract_only command does not identify new interface logic elements. Use the get_ilm_objects command after you are satisfied with the collection of interface logic objects.

8. Use the `write_interface_timing` command to create a timing report for the ILM model. This timing report will be compared to the timing report for the netlist to help verify consistency between the ILM model and the original netlist.

9. Compare the original netlist and ILM interface timing reports by using the `compare_interface_timing` command. This command checks for consistency between the ILM model and the original netlist. For example, the command

   ```
   compare_interface_timing original.wit ilm.wit -output check.cit
   ```

   compares the report for the original netlist in the original.wit file with the report for the ILM model in the ilm.wit file and outputs the comparision report to the check.cit file.

10. Save the model of the block design by using the `write -format ddc` command. Design Compiler supports only the .ddc format for writing out the ILMs.

    By default, the write command saves only the top-level design. Use the `-hierarchy` option to save the entire design. If you do not use the `-output` option to specify the output file name, the `write -format ddc` command creates a file called *top_design*.ddc, where *top_design* is the name of the current design. For more information, see Chapter 5, "Working With Designs in Memory."

    To save an ILM to a file named my_ilm, execute the following command:

    ```
    write -format ddc -hierarchy -output my_ilm
    ```

    Because it is possible to write out an incorrect ILM in the case when the `create_ilm` command did not complete successfully, you should take one of the following precautions before saving the ILM:

    - Check the log file to make sure there are no errors.
    - Check the return status of the `create_ilm` command. If the command returned a 1, then the command succeeded. This is the preferred method of checking because it can be scripted.

Example 9-1 shows the flow for creating ILMs in a multi-scenario environment. When using multi-scenarios, you must create interface timing reports for all your netlists before you create interface timing reports for your scenario ILMs, as shown in Example 9-1.

Example 9-1 Methodology For Using ILMs in a Multi-Scenario Environment

```
compile_ultra
current_scenario sc1
write_interface_timing no_ilm_sc1.wit
current_scenario sc2
write_interface_timing no_ilm_sc2.wit
create_ilm
current_scenario sc1
write_interface_timing ilm_sc1.wit
current_scenario sc2
write_interface_timing ilm_sc2.wit
compare_interface_timing no_ilm_sc1.wit ilm_sc1.wit -output check_sc1.cit
compare_interface_timing no_ilm_sc2.wit ilm_sc2.wit -output check_sc2cit
```

When you create an ILM, Design Compiler automatically performs the following actions:

- Automatically applies the `dont_touch` attribute to the ILM cells.
- Maintains the hierarchy of the original netlist being modeled as an ILM. You can flatten an ILM by ungrouping it using the `ungroup -all -flatten` command.
- 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.
- Handles multi-instantiated designs and ILMs without relying on the `uniquify` command.

When you create an ILM for a block-level that contains multiple instantiations of a hierarchical cell, you do not need to run the `uniquity` command on the subdesign first. Hierarchical cells are automatically uniquified when you run the `compile` command on the subdesign. You can then create an ILM for the mapped block and use that ILM in the top-level design.

At the top level, multi-instantiated ILMs do not need to be uniquified with the `uniquify` command. Note, however, that running the `compile` command at the top level does *not* automatically uniquify any multi-instantiated ILMs.

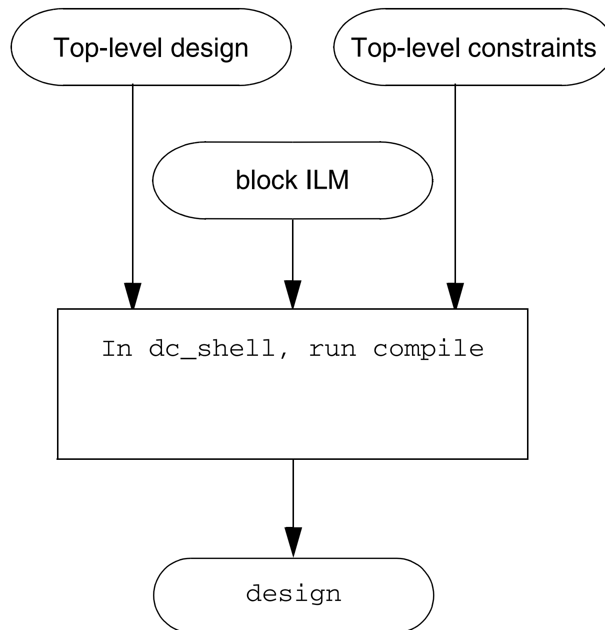
Instantiating and Using ILMs in the Top-Level Design

For ILMs created in Design Compiler topographical mode, the abstract model is saved in the .ddc format without any FRAM view. The tool in the top level treats the entire region over this ILM object as a placement and wiring blockage. The tool does not perform routing over the ILM block.

For ILMs created in IC Compiler, the tool loads the FRAM view for the corresponding ILM. If the FRAM view is not found, the entire region over the ILM is treated as a placement and wiring blockage.

Figure 9-5 shows the flow for instantiating and using ILMs in Design Compiler.

Figure 9-5 Flow for Instantiating and Using Interface Logic Models



To use Design Compiler created ILMs in the top-level design, follow this procedure:

1. Read in the ILM block from a .ddc netlist.
2. Read in the top-level design.
3. Set the current design to the top-level design.
4. Link the design with the top-level design as the current design.
5. Source the top-level constraints.
6. Run the `compile_ultra` command.
7. Use the `write_interface_timing` command to create a timing report for the netlist. This timing report is then compared to the timing report for the ILM model to help verify consistency between the ILM model and the original netlist.
8. Use the `create_ilm` command to generate the interface logic model for the subblock.
9. Use the `write_interface_timing` command to create a timing report for the ILM model. This timing report is then compared to the timing report for the netlist to help verify consistency between the ILM model and the original netlist.

10. Compare the original netlist and ILM interface timing reports by using the `compare_interface_timing` command. This command checks for consistency between the ILM model and the original netlist. For example, the following command

    `compare_interface_timing` *`original.wit ilm.wit`* `-output` *`check.cit`*

    compares the report for the original netlist in the original.wit filein the original.wit file with the report for the ILM model in the ilm.wit file and outputs the comparision report to the check.cit file.

11. Use the `write_file` command to save the design.

To use IC Compiler created ILMs in the top-level design, follow this procedure:

1. Set the link library variable to identify the ILM blocks in the ILM view.

   For example, run
   ```
   set link_library "$link_library block.ILM"
   ```
   Add the Milkyway design library to the Milkyway reference library as follows:
   ```
   set mw_reference_library "$mw_reference_library block.iCC.MW
   ```
2. Perform library setup. For example, run
   ```
   create_mw_design
   open_mw_lib
   ```
3. Read in the top-level design.
4. Set the current design to the top-level design.
5. Use routing (FRAM) views for the ILM blocks for which you want placement and routing resources to be available at the top level. In other words, select the routing (FRAM) views for these blocks by using the `change_macro_view` command.
6. Link the design, with the top-level design as the current design.
7. Source the top-level constraints.
8. Run the `compile_ultra` command.
9. Use the `write_interface_timing` command to create a timing report for the netlist. This timing report is then compared to the timing report for the ILM model to help verify consistency between the ILM model and the original netlist.
10. Use the `create_ilm` command to generate the ILM model for the subblock.
11. Use the `write_interface_timing` command to create a timing report for the ILM model. This timing report will be compared to the timing report for the netlist to help verify consistency between the ILM model and the original netlist.

12. Compare the original netlist and ILM interface timing reports by using the `compare_interface_timing` command. This command checks for consistency between the ILM model and the original netlist. For example, the following command

    `compare_interface_timing` *`original.wit ilm.wit`* `-output` *`check.cit`*

    compares the report for the original netlist in the original.wit file with the report for the ILM model in the ilm.wit file and outputs the comparision report to the check.cit file.

13. Use the `write_file` command to save the design.

Using the create_ilm Command Options

Table 9-1 describes when to use various `create_ilm` options.

Table 9-1 Using the create_ilm Command Options

To do this	Use this option
Only identify interface logic elements, but not build the ILM.	`-identify_only`
Print statistics about the number of design objects in the original design and the model netlist.	`-verbose`
Include the boundary cells of ignored ports.	`-keep_boundary_cells`
Only extract and create an ILM from currently identified interface logic elements.	`-extract_only`
To exclude the fanin and fanout logic from the input and output of ports connected to chip-level nets. For details, see "Controlling the Fanins and Fanouts of Chip-Level Networks" on page 9-6.	`-ignore_ports`
To automatically exclude the fanout logic from input ports connected to a certain percentage of the total number of registers in the design. For details, see "Controlling the Fanins and Fanouts of Chip-Level Networks" on page 9-6.	`-no_auto_ignore`
Specify the number of latch levels for which time borrowing can occur for latch chains that are part of the interface logic. For details, see "Controlling the Number of Latch Levels" on page 9-7.	`-latch_level`
Include all macros. For details, see "Controlling the Logic Included in ILMs" on page 9-5.	`-keep_macros`

Table 9-1 Using the create_ilm Command Options (Continued)

To do this	Use this option
Include interface logic from disabled timing arcs and pins. For details, see "Controlling the Logic Included in ILMs" on page 9-5.	-traverse_disabled_arcs
Include side-load cells of a specified type. For details, see "Controlling Side-Load Cells" on page 9-8.	-include_side_load
Specify that the tool use techniques to reduce the size of the ILM.	-compact

For the complete syntax, see the man page.

Using ILMs with Multicorner-Multimode Designs

ILMs are compatible with multicorner-multimode scenarios. You can apply multicorner-multimode constraints to an ILM and use the ILM in a top-level design. For details, see "Optimizing Multicorner-Multimode Designs in Design Compiler Graphical" in Chapter 10.

Reporting Information About ILMs

Use the following commands to obtain information about ILMs:

- `report_design`

  The `report_design` command reports information that the design is an ILM 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 ILM called ILM_TOP and ILM_TOP is the current design, the report provides the information that it is an ILM.
  - If you have an ILM 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 will report that design A is a subdesign inside the ILM.

- `report_area`

  The `report_area` command provides statistics for both the original netlist (Example 9-2) and the ILM netlist (Example 9-3) to let you know how much reduction occurred for each of the design objects and the total area reduction for ILM designs. However, if instance A is the current design, when you run `report_area` on it, you see only the statistics for the ILM of design A.

- `get_ilm_objects`

  Reports the objects in the current design that are identified as belonging to interface logic.

- `get_ilms`

  Reports the ILM blocks that are defined as part of the current design. You can prevent this command from issuing messages by using the `-quiet` option.

- `create_ilm -verbose`

  Reports the statistics for the original design and the ILM.

- `write_interface_timing` and `compare_interface_timing`

  Use the reports generated by the `write_interface_timing` and `compare_interface_timing` commands to verify consistency between the ILM model and the original netlist. For details, see "Methodology for Creating and Saving ILMs" on page 9-9.

For complete command syntax, see the respective man pages.

The reports in the following examples show that the ILM model has reduced the total cell area from about 790 units to 226 units. Example 9-2 uses the `report_area` command to show the statistics for the original netlist.

Example 9-2 Pre-ILM-Creation Area Report 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 9-3 uses the `report_area` command to show the statistics for the ILM netlist.

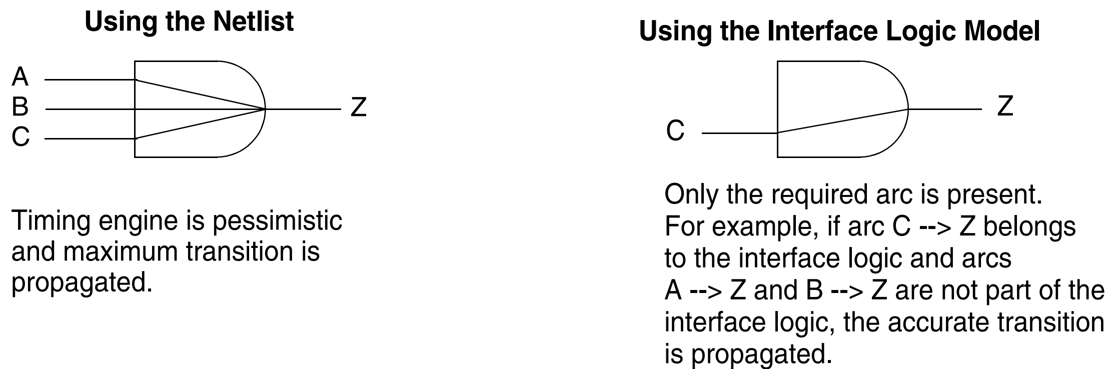
Example 9-3 Post- ILM-Creation Area Report Result When Instance A Is the Current Design

```
****************************************
Report : area
Design : A
Version: 2002.05
Date   : Mon Apr  1 06:48:15 2002
****************************************

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
.
```

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 9-6.

Figure 9-6 Comparison of Timing Propagation Between Netlist and Model



Summary of Commands for ILMs

Table 9-2 lists commands for ILMs.

Table 9-2 Summary of Commands for Interface Logic Models

Command	Description
`create_ilm`	Determines the objects in the current design that are identified as belonging to interface logic, creates an ILM, and saves the model in memory, replacing the original design. The ILM in memory has the same design name as the original block. For option details, see "Using the create_ilm Command Options" on page 9-15.
`get_ilms`	Returns a collection of ILMs defined in the current design. Use the `-quiet` option to prevent this command from issuing messages.
`get_ilm_objects`	Returns a collection of nets, cells, or pins that are identified as belonging to interface logic.
`write_interface_timing`	Verifies consistency between the ILM model and the original netlist when used with the `compare_interface_timing` command. For details, see "Methodology for Creating and Saving ILMs" on page 9-9.

Table 9-2 Summary of Commands for Interface Logic Models (Continued)

Command	Description
`compare_interface_timing`	Verifies consistency between the ILM model and the original netlist when used with the `write_interface_timing` command, see "Methodology for Creating and Saving ILMs" on page 9-9.

For complete command syntax, see the respective man pages.

10

Using Design Compiler Topographical Technology

Topographical technology enables you to accurately predict post-layout timing, area, and power during RTL synthesis without the need for wireload model-based timing approximations. It uses Synopsys' placement and optimization technologies to drive accurate timing prediction within synthesis, ensuring better correlation to the final physical design. This new technology is built in as part of the DC Ultra feature set and is available only by using the `compile_ultra` command in topographical mode.

Design Compiler topographical mode requires a DC Ultra license and a DesignWare license. A Milkyway-Interface license is also necessary if the Milkyway flow is used; this license is automatically included with the DC Ultra license.

This chapter includes the following sections:

- Overview of Topographical Technology
- Starting Design Compiler Topographical Mode
- Inputs and Outputs in Design Compiler Topographical Mode
- Specifying Libraries
- Using Floorplan Physical Constraints
- Performing Automatic High-Fanout Synthesis
- Test Synthesis in Topographical Mode

- Using Power Compiler in Topographical Mode
- Multivoltage Designs
- Compile Flows in Topographical Mode
- Supported Commands, Command Options, and Variables
- Using the Design Compiler Graphical Tool
- Optimizing Multicorner-Multimode Designs in Design Compiler Graphical

Overview of Topographical Technology

In ultra deep submicron designs, interconnect parasitics have a major effect on path delays; accurate estimates of resistance and capacitance are necessary to calculate path delays. In topographical mode, Design Compiler leverages the Synopsys physical implementation solution to derive the “virtual layout” of the design so that the tool can accurately predict and use real net capacitances instead of wireload model-based statistical net approximations. If wireload models are present, they are ignored.

In addition, the tool updates capacitances as synthesis progresses. That is, it considers the variation of net capacitances in the design by adjusting placement-derived net delays based on an updated “virtual layout” at multiple points during synthesis. This approach eliminates the need for overconstraining the design or using optimistic wireload models in synthesis. The accurate prediction of net capacitances drives Design Compiler to generate a netlist that is optimized for all design goals including area, timing, test, and power. It also results in a better starting point for physical implementation.

Topographical technology supports all synthesis flows, including the following:

- Test-ready compile flow (basic scan and DFT MAX adaptive scan)
- Clock-gating flow
- Register retiming

The recommended compile flow in topographical mode is top down as described in the following steps:

1. To use the Design Compiler topographical features, run Design Compiler in topographical mode by entering `dc_shell -topographical` at the command prompt.

   See “Starting Design Compiler Topographical Mode” on page 10-7.

2. Set up the libraries.

   In topographical mode, Design Compiler requires both logical libraries and physical libraries. Design Compiler topographical mode uses the same logical libraries as the Design Compiler wire load mode; it uses the Milkyway format for physical libraries. See“Specifying Libraries” on page 10-9.

3. Read in the design (Verilog, VHDL, netlist, or .ddc). See “Inputs and Outputs in Design Compiler Topographical Mode” on page 10-7.

4. Specify timing, area, power, and test constraints.

   Just as in Design Compiler wireload mode, you set up the design environment by specifying the external operating conditions (manufacturing process, temperature, and voltage), loads, drives, fanouts, and so on as described in Chapter 6, “Defining the

Design Environment." However, you do not specify the wire load mode (Design Compiler ignores the wire load mode if you set it). You also specify timing and area goals as described in Chapter 7, "Defining Design Constraints."

In addition, you can specify power constraints as described in "Using Power Compiler in Topographical Mode" on page 10-69 and the test configuration as described in "Test Synthesis in Topographical Mode" on page 10-68. Topographical technology supports multivoltage designs, described in "Multivoltage Designs" on page 10-71.

5. (Optional) Provide floorplan information.

   The principal reason for using floorplan constraints in topographical mode is to accurately represent the placement area and to improve timing correlation with the post-place-and-route design. You can provide high-level physical constraints that determine core area and shape, port location, macro location and orientation, voltage areas, placement blockages, and placement bounds. These physical constraints can be derived from IC Compiler floorplan data, extracted from an existing Design Exchange Format (DEF) file, or created manually. See "Using Floorplan Physical Constraints" on page 10-15.

   Sometimes Design Compiler topographical and IC Compiler have different environment settings. These differences can lead to correlation problems. To help fix correlation issues between Design Compiler topographical and IC Compiler, use the `write_environment` command. For details, see "Comparing Design Compiler Topographical and IC Compiler Environments" on page 12-4.

6. (Optional) Visually verify the floorplan.

   Use the Design Vision layout window to visually verify that your pre-synthesis floorplan is laid out according to your expectations. The layout view automatically displays floorplan constraints read in with the `extract_physical_constraints` command or read in with Tcl commands. You need to link all applicable designs and libraries to obtain an accurate floorplan. For details, see the *Design Vision User Guide* or Design Vision Online Help.

7. Check the design.

   Before synthesizing your design, check that all designs and libraries have the necessary data for compile to run successfully by using the `-check_only` option of the `compile_ultra` command. The `-check_only` option quickly reports potential problems that could cause the tool to stop during `compile_ultra` or to produce unsatisfactory correlation with your physical implementation. Use the `-check_only` option to help debug such problems as

   - Missing TLUPlus files and missing physical library cells
   - Multiple logical library cells with the same names
   - Discrepancies in technology data between multiple physical libraries
   - Missing placement location for cells and ports of ILMs or physical hierarchical modules

- Missing `dont_touch` attributes for ILMs or physical hierarchical modules
- Missing floor plan information such as core area constraints (through site row definition), port location, macro location, ILM location, and physical hierarchy location.

Important:
Always execute the `compile_ultra` optimization step with the same set of options that were used when you executed `compile_ultra -check_only`.

8. Compile the design by using the `compile_ultra -gate_clock -scan` command.

   When you use the `-gate_clock` option, clock gating is implemented in the design according to options set by the `set_clock_gating_style` command. For more information on this command, see the *Power Compiler User Guide*.

   When you use the `-scan` option, the tool performs test-ready compile. During test-ready compile, the tool employs the scan insertion design-for-test technique, which replaces regular flip-flops with flip-flops that contain logic for testability. Test-ready compile reduces iterations and design time, by accounting for the impact of the scan implementation during the logic optimization process. For optimization details, see *Design Compiler Optimization Reference Manual*.

9. Insert scan chains.

   Run the `insert_dft` command to insert scan chains. For more information on this command, see the *DFT Compiler User Guide*.

10. Perform an incremental compile by using the `compile_ultra -incremental -scan`.

    The main goal for `compile_ultra -incremental` is to enable topographical-based optimization for post-topographical-based synthesis flows such as retiming, design-for-test (DFT), DFT MAX, and minor netlist edits. See "Performing an Incremental Compile" on page 10-73.

11. (Optional) Visually inspect the floorplan and placement results.

    Use the Design Vision layout window to verify your floorplan and placement results. You can

    - Examine the placement and orientation of objects such as macro cells, port locations, and physical constraints
    - Examine the placement of critical timing path objects
    - Analyze floorplan-related congestion and identify the causes of congestion hotspots

    For details, see the *Design Vision User Guide* or the Design Vision Online Help.

12. Write out the design. Formats include .ddc, Milkyway, or Verilog. See "Inputs and Outputs in Design Compiler Topographical Mode" on page 10-7.

The scripts shown in Example 10-1 and Example 10-2 suggest how to set up your libraries and use Design Compiler topographical technology to synthesize your design.

Example 10-1 Default Flow for Topographical Synthesis

```
dc_shell -topo

set search_path "search_path ./libraries"
set link_library "* max_lib.db"
set target_library "max_lib.db"

create_mw_lib -technology $mw_tech_file -mw_reference_library \
                $mw_reference_library $mw_lib_name
open_mw_lib $mw_lib_name

read_verilog rtl.v
source top_constraint.sdc
compile_ultra
report_timing
change_names -rules verilog -h
write -format ddc -hierarchy -output top_synthesized.ddc
write -f verilog -h -o netlist.v
write_sdf sdf_file_name
write_parasitics -o parasitics_file_name
write_sdc sdc_file_name
```

Example 10-2 Flow With Physical Constraints

```
dc_shell -topo

set search_path "search_path ./libraries"
set link_library "* max_lib.db"
set target_library "max_lib.db"

read_verilog rtl.v
source top_constraint.sdc
extract_physical_contraints def_file_name
compile_ultra
report_timing
change_names -rules verilog -h
write -format ddc -hierarchy -output top_synthesized.ddc
write -f verilog -h -o netlist.v
write_sdf sdf_file_name
write_parasitics -o parasitics_file_name
write_sdc sdc_file_name
write_physical_constraints -o phys_cstr_file_name.tcl
```

Starting Design Compiler Topographical Mode

To use the Design Compiler topographical features, you must run dc_shell in topographical mode. At the prompt, enter

```
dc_shell -topographical
```

In topographical mode, the dc_shell command-line prompt appears as

```
dc_shell-topo>
```

To query for topographical mode, run the `shell_is_in_topographical_mode` command. The command returns 1 if the topographical mode is active; otherwise it returns 0. Topographical technology is Tcl based. When you run the `compile_ultra` command in this mode, the Design Compiler topographical features are automatically used. All `compile_ultra` command options are supported.

Note:
Design Compiler in topographical mode supports only a subset of dc_shell commands, command options, and variables. For more information, see “Supported Commands, Command Options, and Variables.”

Inputs and Outputs in Design Compiler Topographical Mode

Figure 10-1 shows the inputs and outputs in Design Compiler topographical mode. The sections that follow provide more information.

Figure 10-1 Inputs and Outputs in Design Compiler Topographical Mode

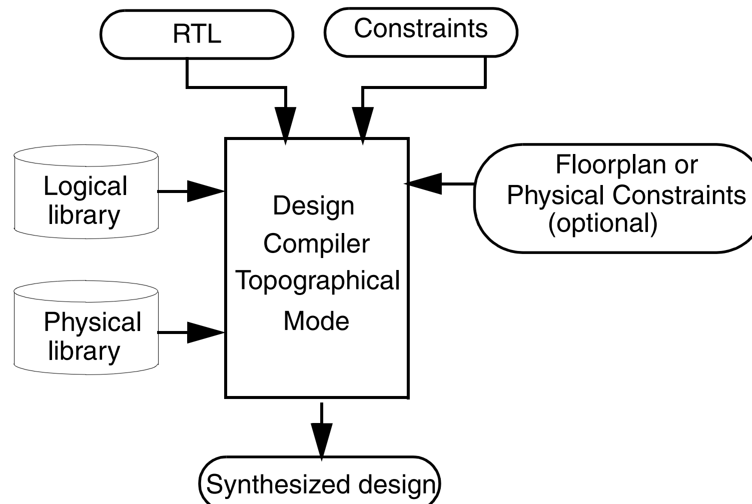


Table 10-1 and Table 10-2 describe the inputs and outputs in topographical mode.

Table 10-1 Inputs in Design Compiler Topographical Mode

Input	Description
Design	RTL or gate-level netlist.
Constraints	Timing and optimization constraints. Note that if you specify wireload models, Design Compiler ignores them.
Logical library	Liberty format (.lib or .db). For more information, see “Specifying Logical Libraries” on page 10-9.
Physical library	Milkyway format. For more information, see “Specifying Physical Libraries” on page 10-10. Note: The .pdb format is not supported.
Floorplan or physical constraints	High-level physical constraints that determine items such as core area and shape, port location, macro location and orientation, voltage areas, placement blockages, and placement bounds. For more information, see “Using Floorplan Physical Constraints” on page 10-15.

Table 10-2 Outputs in Design Compiler Topographical Mode

Output	Description
.ddc	This format contains back-annotated net delays and constraints. Subsequent topographical mode sessions restore virtual layout data. The .ddc format is recommended for subsequent topographical mode optimizations and verification.
Milkyway	This format contains back-annotated net delays and constraints. The Milkyway format cannot be read back into topographical mode for subsequent optimizations. This format is recommended if you intend to use Synopsys tools for the backend flow.
ASCII	This format does not contain back-annotated delays or Synopsys Design Constraints (SDC). Use the `write_sdc` command to write out the SDC and the `write_parasitics` command to write out parasitics. This format is recommended only if you intend to use a third-party tool.

Note:
You cannot use the Milkyway format to store design data for unmapped designs or non-uniquified designs. Before you use the `write_milkyway` command, run the following command:

```
uniquify -force -dont_skip_empty_designs
```

Specifying Libraries

In topographical mode, Design Compiler requires both logical libraries and physical libraries. Topographical mode also supports synthesis with black boxes. For more details about black box support, see "Support for Black Boxes" on page 10-14.

Note:
Occasionally, IC Compiler adds support for new technology file attributes in the physical libraries that are not yet supported in Design Compiler. In these cases, a TFCHK-009 error message is issued in Design Compiler but not in IC Compiler when using the same technology file. If the new attributes are safe to ignore, you can set the `ignore_tf_error` variable to true, which enables the tool to ignore the unsupported attributes. For details, see "Fixing Errors Caused by New Unsupported Technology File Attributes" on page 12-3.

Specifying Logical Libraries

Design Compiler topographical mode uses the same logical libraries as the Design Compiler wire load mode. Use the following commands to set up logical libraries:

```
set search_path "search_path ./libraries"
set link_library "* max_lib.db"
set target_library "max_lib.db"
```

For more information on logical libraries, see "Working With Libraries" on page 4-1.

Note:
Because multivoltage designs use power domains, these designs usually require that certain library cells are marked as always-on cells and certain library cell pins are marked as always-on library cell pins. These always-on attributes are necessary to establish any always-on relationships between power domains. For more information, see the *Power Compiler User Guide*.

Specifying Physical Libraries

You use the Milkyway design library to specify physical libraries and save designs in Milkyway format. The inputs required to create a Milkyway design library are the Milkyway reference library and the Milkyway technology file.

The Milkyway reference library contains the physical representation of standard cells and macros. In topographical mode, the Milkyway reference library uses the FRAM abstract view to store information. The reference library also defines the placement unit tile (the width and height of the smallest placeable instance and the routing directions).

The Milkyway technology file (.tf), contains technology-specific information required to route a design. Design Compiler will automatically derive routing layer directions if your Milkyway library file is missing this information. Derived routing layer directions are saved in the .ddc file. You can override the derived routing layer direction by using the `set_preferred_routing_direction` command. To report all routing directions, use the `report_preferred_routing_direction` command.

The following steps provide an overview of how to create a Milkyway design library; for more information on the Milkyway database, see Chapter 11, “Using a Milkyway Database and the Milkyway documentation.

1. Define the power and ground nets. For example, set the following variables:

   ```
   set mv_power_net VDD
   set mw_ground_net VSS
   set mw_logic1_net VDD
   set mw_logic0_net VSS
   set mw_power_port VDD
   set mw_ground_port VSS
   ```

   If you do not set these variables, power and ground connections are not made during execution of `write_milkyway`. Instead power and ground nets can get translated to signal nets.

2. Use the `create_mw_lib` command to create the Milkyway design library. For example,

   ```
   create_mw_lib -technology $mw_tech_file \
         -mw_reference_library $mw_reference_library
                    $mw_design_library_name
   ```

3. Use the `open_mw_lib` command to open the Milkyway library that you created. For example, enter

   ```
   open_mw_lib $mw_design_library_name
   ```

4. (Optional) Use the `set_tlu_plus` command to attach TLUPlus files. For example,

   ```
   set_tlu_plus_files-max_tluplus $max_tlu_file \
   ```

```
                       -min_tluplus $min_tlu_file\
                       -tech2itf_map $prs_map_file
```

For more information, see “Using TLUPlus for RC Estimation” on page 10-13.

5. In subsequent topographical mode sessions, you use the `open_mw_lib` command to open the Milkyway library. If you are using TLUPlus files for RC estimation, use the `set_tlu_plus_files` command to attach these files. For example,

```
open_mw_lib $mw_design_library_name
set_tlu_plus_files-max_tluplus $max_tlu_file \
                       -min_tluplus $min_tlu_file\
                       -tech2itf_map $prs_map_file
```

The following Milkyway library commands are also supported:

- `copy_mw_lib`
- `close_mw_lib`
- `report_mw_lib`
- `current_mw_lib`
- `check_tlu_plus_files`
- `write_mw_lib_files`
- `set_mw_lib_references`

Verifying Library Consistency

Consistency between the logic library and the physical library is critical to achieving good results. Before you process your design, ensure that your libraries are consistent by running the `check_library` command.

```
dc_shell-topo> check_library
```

The `check_library` command provides the following capabilities:

- Checks the integrity of individual logical and physical libraries.
- Checks consistency between logical libraries.
- Checks consistency between logical libraries and physical libraries.
- Checks consistency between physical libraries and technology files.

By default, the check_library command performs consistency checks between the logic libraries specified in the link_library variable and the physical libraries in the current Milkyway design library. You can also explicitly specify logic libraries (by using the -logic_library_name option) or Milkyway reference libraries (by using the -mw_library_name option). If you explicitly specify libraries, these override the default libraries.

By default, the check_library command performs the following consistency checks between the logic library and the physical library:

- Verify that all cells exist in both the logic library and the physical library, including any physical-only cells.
- Verify that the pins on each cell are consistent between the logic library and the physical library.

  This validation includes consistency in naming, direction, and type (including power and ground pins).

  Note:
  If the logic library does not contain pg_pin definitions, Design Compiler uses the power and ground pins as defined in the physical library.

You can also perform the following consistency checks, by setting the appropriate options with the set_check_library_options command:

- Verify that the area for each cell (except pad cells) is consistent between the logic library and the physical library.

  ```
  dc_shell-topo> set_check_library_options -cell_area
  ```

- Verify that the cell footprints are consistent between the logic library and the physical library.

  ```
  dc_shell-topo> set_check_library_options -cell_footprint
  ```

- Verify that the same bus naming style is used in the logic library and the physical library.

  ```
  dc_shell-topo> set_check_library_options -bus_delimiter
  ```

Specify the option for each check that you would like to enable, or to enable all consistency checks, use the -logic_vs_physical option.

To reset the library checks to the default settings (cell name and pin consistency checks), run the set_check_library_options -reset command.

To see the enabled library consistency library checks, run the report_check_library_options -logic_vs_physical command.

If the check_library command reports any inconsistencies, you must fix these inconsistencies before you process your design. For more information about logic libraries, see the Library Compiler documentation. For more information about physical libraries, see the Milkyway documentation.

Using TLUPlus for RC Estimation

TLUPlus files contain resistance and capacitance look-up tables and model ultra deep submicron (UDSM) process effects. If TLUPlus are available or are used in your backend flow, it is recommended that you use them for RC estimation in Design Compiler topographical mode.

TLUPlus files provide more accurate capacitance and resistance data, thereby improving correlation with back-end results.

You use the set_tlu_plus_files command to specify TLUPlus files. In addition, use the -tech2itf_map option to specify a map file, which maps layer names between the Milkyway technology file and the process Interconnect Technology Format (ITF) file. For example,

```
set_tlu_plus_files -max_tluplus $max_tlu_file \
                   -min_tluplus $min_tlu_file\
                   -tech2itf_map $prs_map_file
```

You can use the check_tlu_plus_files command to check TLUPlus settings.

For more information on the map file, see the Milkyway documentation. To ensure that you are using the TLUPlus files, check the compile_ultra log for the following message:

```
********************************************************
Information: TLU Plus based RC computation is enabled.
(RCEX-141)
********************************************************
```

Topographical mode supports the extract_rc command to perform 2.5D extraction. The command calculates delays based on the Elmore delay model and can update back-annotated delay and capacitance numbers on nets. Use this command after the netlist has been edited. If you used the set_tlu_plus_files command to specify the TLUPlus technology files, the tool performs extraction based on TLUPlus technology. Otherwise, the tool performs extraction using the extraction parameters in your physical library. Use the set_extraction_options command to specify the parameters that influence extraction and the report_extraction_options command to report the parameters that influence the post-route extraction.

Support for Black Boxes

Topographical mode supports synthesis with black boxes. The tool can create physical library cells for the following:

- Logical library cells (leaf cells and macros)
- Empty hierarchy cells or black-boxed modules
- Unlinked or unresolved cells
- Unmapped cells

The tool issues the following warning message when it creates physical library cells:

```
Warning: Created physical library cell for logical library
%s. (OPT-1413)
```

Using Floorplan Physical Constraints

This section describes how to create, import, reset, save, and report physical constraints in the following subsections:

- Physical Constraints Overview
- Extracting Physical Constraints From IC Compiler Using the write_def Command
- Importing Physical Constraints From an IC Compiler DEF Formatted File
- Exporting Physical Constraints From IC Compiler Using the write_floorplan Command
- Importing Physical Constraints From an IC Compiler write_floorplan Formatted File
- Manually Defining Physical Constraints
- Specifying Relative Placement
- Placement Options: Magnet Placement
- Resetting Physical Constraints
- Saving Physical Constraints Using the write_floorplan Command
- Reporting Physical Constraints

Physical Constraints Overview

The principal reason for using floorplan constraints in topographical mode is to improve timing correlation with the post-place-and-route tools, such as IC Compiler, by considering floorplanning information during optimizations. DC Ultra topographical mode supports high-level physical constraints such as die area, core area and shape, port location, macro location and orientation, voltage areas, placement blockages, wiring keepouts, pre-routes, and placement bounds.

You provide physical information to Design Compiler topographical mode in the following ways:

- You can export the physical data from IC Compiler by using the `write_def` command within IC Compiler and can import this data into Design Compiler by using the `extract_physical_constraints` command.

  See "Extracting Physical Constraints From IC Compiler Using the write_def Command" on page 10-16 and "Importing Physical Constraints From an IC Compiler DEF Formatted File" on page 10-17.

- You can export the physical data from IC Compiler by using the `write_floorplan` command within IC Compiler and import this data into Design Compiler by using the `read_floorplan` command.

  See “Exporting Physical Constraints From IC Compiler Using the write_floorplan Command” on page 10-26 and “Importing Physical Constraints From an IC Compiler write_floorplan Formatted File” on page 10-26.

- You can create physical constraints manually. See “Manually Defining Physical Constraints” on page 10-29.

If you do not provide any physical constraints from a floorplanning tool, Design Compiler uses the following default physical constraints:

- Aspect ratio of 1.0 (that is, a square placement area)
- Utilization of 0.6 (that is, forty percent of empty space in the core area)

Extracting Physical Constraints From IC Compiler Using the write_def Command

To improve timing, area, and power correlation between Design Compiler and IC Compiler, you can read your mapped Design Compiler netlist into IC Compiler, create a basic floorplan in IC Compiler, export this floorplan from IC Compiler, and read the floorplan back into Design Compiler.

To export floorplan information from IC Compiler for use in DC Ultra topographical mode, you can use the `write_def` command in IC Compiler. The `write_def` command exports physical design data and writes this data to a Design Exchange Format (DEF) file which you can read into DC Ultra topographical mode. Example 10-3 uses the `write_def` command to write physical design data to the my_physical_data.def file.

Example 10-3 Using the write_def Command to Extract Physical Data From IC Compiler

```
icc_shell> write_def -version 5.7 -rows_tracks_gcells -macro -pins \
     -blockages -specialnets -vias -regions_groups -verbose \
     -output my_physical_data.def
```

For details about the `write_def` command, see the man page.

Importing Physical Constraints From an IC Compiler DEF Formatted File

This section describes the imported DEF physical constraints, incremental DEF extraction, and port and macro name matching considerations, in the following subsections:

- Importing a DEF Floorplan Overview
- Imported DEF Constraints
- Importing Incremental DEF Information Using the extract_physical_constraints Command
- Matching Names of Macros and Ports

Importing a DEF Floorplan Overview

To import floorplan information from a DEF file, use the `extract_physical_constraints` command. This command imports physical information from the specified DEF file and applies these constraints to the design. The applied constraints are saved in the .ddc file and do not need to be reapplied in a new topographical mode session when you read in the .ddc.

The following command shows how you can import physical constraints from multiple DEF files:

```
dc_shell-topo> extract_physical_constraints \
                    {des_1.def des2.def … des_N.def}
```

For information on incremental extraction from DEF files, see "Importing Incremental DEF Information Using the extract_physical_constraints Command" on page 10-24.

Note:
When you use the `extract_physical_constraints` command to read a DEF file, DC Ultra topographical mode automatically resolves the site name in the floorplan to match the name of the tile in the Milkyway reference libraries. Milkyway reference libraries usually have the site name set to unit by default. However, you might still need to use the `mw_site_name_mapping` variable to define the name mappings if the site dimension does not match unit or if there is more than one site type being used in the DEF. For example,

```
set mw_site_name_mapping def_site_name milkyway_site
```

Multiple pairs of values can be specified for the `mw_site_name_mapping` variable.

Imported DEF Constraints

This section describes the physical constraints imported from the DEF file with the `extract_physical_constraints` command, in the following subsections:

- Die Area
- Placement Area
- Macro Location and Orientation
- Hard, Soft, and Partial Placement Blockages
- Wiring Keepouts
- Placement Bounds
- Port Location
- Preroutes
- Site Array Information

To visually inspect your extracted physical constraints, use the layout view in the Design Vision layout window. All physical constraints extracted from the DEF file are automatically added to the layout view.

Note:
Voltage areas are not defined in a DEF file. Therefore, for multivoltage designs, you have to use the `create_voltage_area` command to define voltage areas for the tool. If you are using IC Compiler as your floorplanning tool, you can use the `write_floorplan` and `read_floorplan` commands to obtain floorplan information that automatically includes voltage areas. You do not need to define voltage areas manually when using these commands.

Die Area

The DC Ultra topographical mode command, `extract_physical_constraints`, supports automatic die area extraction from the DEF file. DEF files are generated by floorplanning tools and used by DC Ultra to import physical constraints. The tool can extract both rectangular and rectilinear die areas that are defined in the DEF file. Keep in mind that the die area is also known as the cell boundary. The die area represents the silicon boundary of a chip and encloses all objects of a design, such as pads, I/O pins, and cells.

The following example shows a die area definition in a DEF file:

```
DEF
   UNITS DISTANCE MICRONS 1000 ;
   DIEAREA ( 0 0 ) ( 0 60000 ) ( 39680 60000 ) ( 39680 40000 ) \
```

```
( 59360 40000 ) ( 59360 0 ) ;
```

The example below shows how DC Ultra, in topographical mode, would translate thise DEF definition into Tcl when you write out your physical constraints using the `write_physical_constraints` command.

```
create_die_area -polygon { { 0.000 0.000 } { 0.000 60.000 }\
       { 39.680 60.000 } { 39.680 40.000 } { 59.360 40.000 } \
       { 59.360 0.000 } { 0.000 0.000 } }
```

Placement Area

Placement area is computed as the rectangular bounding box of the site rows.

Macro Location and Orientation

When you use the `extract_physical_constraints` command, for each cell with a location and the FIXED attribute specified in the DEF, Design Compiler sets the location on the corresponding cell in the design. Example 10-4 shows DEF macro location and orientation information.

Note:
E = east rotation and W = west rotation

Example 10-4 DEF Macro Location and Orientation Information

```
COMPONENTS 2 ;
   - macro_cell_abx2  + FIXED ( 4350720 8160 ) E ;
   - macro_cell_cdy1 + FIXED ( 4800 8160 ) W ;
END COMPONENTS
```

The Tcl equivalent commands are shown in Example 10-5.

Example 10-5 Tcl Equivalent Macro Location and Orientation Information

```
set_cell_location macro_cell_abx2 -coordinate { 4350.720 8.160 } -orientation E -fixed
set_cell_location macro_cell_cdy1 -coordinate { 4.800 8.160 } -orientation W -fixed
```

Hard, Soft, and Partial Placement Blockages

The `extract_physical_constraints` command can import hard, soft, and partial placement blockages defined in the DEF file.

Note:
DEF versions prior to version 5.7 did not support partial blockages. In addition, if your floorplanning tool creates a DEF file with DEF version 5.6, you need to manually add the #SNPS_SOFT_BLOCKAGE pragma to specify a soft blockage, as shown in Example 10-9.

Example 10-6 shows DEF hard placement blockage information.

Example 10-6 DEF Hard Placement Blockage Information

```
BLOCKAGES 50 ;
...
   - PLACEMENT RECT ( 970460 7500 ) ( 3247660 129940 )
...
END BLOCKAGES
```

The Tcl equivalent command is shown in Example 10-7.

Example 10-7 Tcl Equivalent Hard Placement Blockage Information

```
create_placement_blockage -name def_obstruction_23
                    -bbox { 970.460 7.500 3247.660 129.940 }
```

For a soft placement blockage, if your extracted DEF information is as shown in Example 10-8 (DEF version 5.7) or Example 10-9 (DEF version 5.6),

Example 10-8 DEF Version 5.7 Soft Placement Blockage Information

```
BLOCKAGES 50 ;
...
- PLACEMENT + SOFT RECT ( 970460 7500 ) ( 3247660 129940 ) ;
...
END BLOCKAGES
```

Example 10-9 DEF Version 5.6 Soft Placement Blockage Information

```
BLOCKAGES 50 ;
...
- PLACEMENT RECT ( 970460 7500 ) ( 3247660 129940 ) ; #SNPS_SOFT_BLOCKAGE
...
END BLOCKAGES
```

then the Tcl equivalent command is shown in Example 10-10.

Example 10-10 Tcl Equivalent Soft Placement Blockage Information

```
create_placement_blockage -name def_obstruction_23 \
                    -bbox { 970.460 7.500 3247.660 129.940 } \
                    -type soft
```

For a partial placement blockage, if your extracted DEF information is as shown in Example 10-11,

Example 10-11 DEF Partial Placement Blockage Information

```
BLOCKAGES 50 ;
...
- PLACEMENT + PARTIAL 80 RECT ( 970460 7500 ) ( 3247660 129940 ) ;
...
END BLOCKAGES
```

then the Tcl equivalent command is shown in Example 10-12.

Example 10-12 Tcl Equivalent Partial Placement Blockage Information

```
create_placement_blockage -name def_obstruction_23 \
                      -bbox { 970.460 7.500 3247.660 129.940 } \
                      -type partial
                      -blocked_percentage 80
```

Wiring Keepouts

For wiring keepouts defined in the DEF, Design Compiler creates wiring keepouts on the design.

Example 10-13 shows DEF wiring keepout information.

Example 10-13 DEF Wiring Keepout Information

```
BLOCKAGES 30 ;
...
   - LAYER METAL6 RECT ( 0 495720 ) ( 4050 1419120 );
...
END BLOCKAGES
```

Placement Bounds

If REGIONS defining bounds exist in the DEF, `extract_physical_constraints` imports placement bounds. Also, if there are cells in the related GROUP attached to the region, these cells are fuzzy matched with the ones in the design, and the matched cells are attached to the bounds in the following two ways:

- If there are regions in the design with the same name as in the DEF, the cells in the related group are attached to the region by the `update_bounds` command in incremental mode.
- If the region does not exist in the design, it is created with the same name as in the DEF file by applying the `create_bound` command; matched cells in the related group are also attached.

Example 10-14 shows imported placement bounds information.

Example 10-14 DEF Placement Bounds Information

```
REGIONS 1 ;
 - c20_group ( 201970 40040 ) ( 237914 75984 ) + TYPE FENCE ;
END REGIONS

GROUPS 1 ;
 - c20_group
     cell_abc1
     cell_sm1
```

```
     cell_sm2
    + SOFT
    + REGION c20_group ;
END GROUPS
```

The Tcl equivalent commands are shown in Example 10-15.

Example 10-15 Tcl Equivalent Placement Bounds Information

```
create_bounds \
  -name "c20_group" \
  -coordinate {201970 40040 237914 75984} \
  -exclusive \
  {cell_abc1 cell_sm1 cell_sm2}
```

Port Location

When you use the `extract_physical_constraints` command, for each port with the location specified in the DEF, Design Compiler sets the location on the corresponding port in the design.

Example 10-16 shows imported port location information.

Example 10-16 DEF Port Information

```
PINS 2 ;
   -Out1 + NET Out1 + DIRECTION OUTPUT + USE SIGNAL +
      LAYER M3 (0 0) (4200 200) + PLACED (80875 0) N;

   -Sel0 + NET Sel0 + DIRECTION INPUT + USE SIGNAL +
      LAYER M4 (0 0)(200 200) + PLACED (135920 42475) N;
END PINS
```

The Tcl equivalent commands are shown in Example 10-17.

Example 10-17 Tcl Equivalent Port Information

```
set_port_location Out1 -coordinate {80.875 0.000}
         -layer_name M3 -layer_area {0.000 0.000 4.200 0.200}

set_port_location Sel0 -coordinate {135.920 0.000}
         -layer_name M4 -layer_area {0.000 0.000 0.200 0.200}
```

Ports with changed names and multiple layers are supported. Example 10-18 shows DEF information for such a case.

Example 10-18 DEF Port Information

```
PINS 2 ;
   - sys_addr\[23\].extra2 + NET sys_addr[23] + DIRECTION INPUT + USE SIGNAL +
      LAYER METAL4 ( 0 0 ) ( 820 5820 ) + FIXED ( 1587825 2744180 ) N ;
   - sys_addr[23] + NET sys_addr[23] + DIRECTION INPUT + USE SIGNAL + LAYER
```

```
      METAL3 ( 0 0 ) ( 820 5820 ) + FIXED ( 1587825 2744180 ) N ;
END PINS
```

The corresponding Tcl commands are shown in Example 10-19.

Example 10-19 Tcl Equivalent Port Information

```
set_port_location sys_addr[23] -coordinate { 1587.825 2744.180 } \
 -layer_name METAL3 -layer_area {0.000 0.000 0.820 5.820}
set_port_location sys_addr[23] -coordinate { 1587.825 2744.180 }\
 -layer_name
METAL4 -layer_area {0.000 0.000 0.820 5.820} -append
```

Port orientation is also supported. Example 10-20 shows DEF information for such a case.

Example 10-20 DEF Port Information

```
PINS 1;
   - OUT + NET OUT + DIRECTION INPUT + USE SIGNAL
      + LAYER m4 ( -120 0 ) ( 120 240 )
      + FIXED ( 4557120 1726080 ) S ;
END PINS
```

The corresponding Tcl commands are shown in Example 10-21.

Example 10-21 Tcl Equivalent Port Information

```
set_port_location OUT -coordinate { 4557.120 1726.080 } -layer_name m4 \
                     -layer_area {-0.120 -0.240 0.120 0.000}
```

Preroutes

Design Compiler extracts preroutes that are defined in the DEF file.

Example 10-22 shows imported preroute information.

Example 10-22 DEF Preroute Information

```
SPECIALNETS 2 ;
- vdd
 + ROUTED METAL3 10000 + SHAPE STRIPE ( 10000 150000 ) ( 50000 * )
 + USE POWER ;
...
END SPECIALNETS
```

The Tcl equivalent commands are shown in Example 10-23.

Example 10-23 Tcl Equivalent Preroute Information

```
create_net_shape -no_snap -type path -net vdd -datatype 0 -path_type 0 \
         -route_type pg_strap -layer METAL3 -width 10.000 \
         -points {{10.000 150.000} {50.000 150.000}}
```

Site Array Information

Design Compiler imports site array information that is defined in the DEF file. Site arrays in the DEF file define the placement area.

Example 10-24 shows imported site array information.

Example 10-24 DEF Site Array Information

```
ROW ROW_0 core 0 0 N DO 838 BY 1 STEP 560 0;
```

The Tcl equivalent commands are shown in Example 10-25.

Example 10-25 Tcl Equivalent Site Array Information

```
create_site_row -name ROW_0 -coordinate {0.000 0.000}\
               -kind core -orient 0 -dir H -space 0.560 -count 838
```

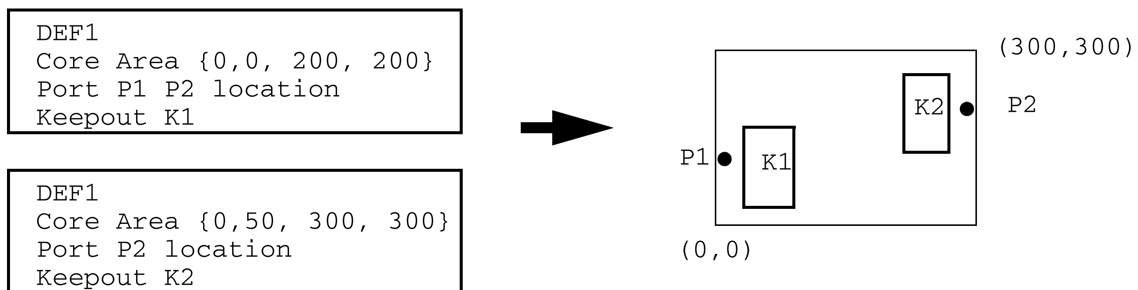
Importing Incremental DEF Information Using the extract_physical_constraints Command

By default, the `extract_physical_constraints` command runs in incremental mode. That is, if you use the command to process multiple DEF files, the command preserves existing physical annotations on the design. In incremental mode, the placement area is imported based on the current core area and site rows in the DEF whereas, in non-incremental mode, the placement area is imported based on the site rows in the DEF. Conflicts are resolved as follows:

- Physical constraints that can have only one value are overwritten by the value from the latest DEF file. That is, port location and macro location are overwritten.
- Physical constraints that can have accumulated values are recomputed. That is, core area can be recomputed based on the existing value and the site row definitions in the latest DEF file. Placement keepouts from different DEF files are accumulated and the final keepout geometry is computed internally during synthesis.

Figure 10-2 shows an example of incremental extraction performed by the `extract_physical_constraints` command.

Figure 10-2 Incremental Extraction with the extract_physical_constraints command




```
extract_physical_constraints DEF1.def DEF2.def
```

To disable incremental mode, use the `-no_incremental` option of the `extract_physical_constraints` command.

Matching Names of Macros and Ports

By default, when the `extract_physical_constraints` command applies physical constraints in topographical mode, it uses an intelligent name matching algorithm to match macros and ports in the DEF file with macros and ports in memory. The command uses the intelligent name matching capability when it does not find an exact match.

The `extract_physical_constraints` command reads the DEF files generated from a netlist that could have different object names than the netlist in memory. These name mismatches can be caused by automatic ungrouping and the `change_names` command. Typically, hierarchy separators and bus notations are sources of these mismatches.

For example, automatic ungrouping by the `compile_ultra` command followed by the `change_names` command might result in the forward slash (/) separator being replaced with an underscore (_) character. Therefore, a macro named *a/b/c/macro_name* in the RTL might be named *a/b_c_macro_name* in the mapped netlist, which is the input to the back-end tool. When extracting physical constraints from the DEF file, the `extract_physical_constraints` command automatically resolves these name differences by using an intelligent name matching algorithm.

To disable intelligent name matching, you can use the `-exact` option of the `extract_physical_constraints` command. This option allows you to specify that the objects in the netlist in memory be matched exactly with the corresponding objects in the DEF file. When you use the `-verbose` option of the `extract_physical_constraints` command, the tool displays an informational message:

```
Information: Fuzzy match cell %s in netlist with instance
%s in DEF.
```

By default, the following characters are considered equivalent:

- Hierarchical separators { / _ . }

  For example, a cell named *a.b_c/d_e* is automatically matched with the string *a/b_c.d/e* in the DEF file.

- Bus notations { [ ] __ ( ) }

  For example, a cell named *a [4] [5]* is automatically matched with the string *a_4__5_* in the DEF file.

To define the rules used by the intelligent name matching algorithm, use the `set_fuzzy_query_options` command. For more information, see the *Design Compiler Command-Line Interface Guide*.

Exporting Physical Constraints From IC Compiler Using the write_floorplan Command

To improve timing, area, and power correlation between Design Compiler and IC Compiler, you can read your mapped Design Compiler netlist into IC Compiler, create a basic floorplan in IC Compiler, export this floorplan from IC Compiler, and read the floorplan back into Design Compiler.

To export floorplan information from IC Compiler for use in DC Ultra topographical mode, you can use the `write_floorplan` command in IC Compiler. The `write_floorplan` command writes a Tcl script that you read into DC Ultra topographical mode to re-create elements of the floorplan of the specified design. Example 10-26 uses the `write_floorplan` command to write out all placed standard cells to a file called placed_std.fp.

Example 10-26 Using write_floorplan to Export Physical Data From IC Compiler

```
icc_shell> write_floorplan -placement {io hard_macro soft_macro}\
                           -create_terminal -row -create_bound
                           -preroute floorplan_for_DC.fp
```

For details about the `write_floorplan` command, see the man page.

Importing Physical Constraints From an IC Compiler write_floorplan Formatted File

Use the `read_floorplan` command to read in a script file generated by the `write_floorplan` command that contains commands describing floorplan information. You can also use the `source` command to import the floorplan information. However, the `source` command reports errors and warnings that are not applicable to DC Ultra in topographical

mode. The `read_floorplan` command removes these unnecessary errors and warnings. In addition, you need to enable fuzzy name matching manually when you use the `source` command. The `read_floorplan` command automatically enables fuzzy name matching.

To visually inspect your imported physical constraints, use the layout view in the Design Vision layout window.

Imported physical constraints are automatically saved to your .ddc file. To save the physical constraints in a separate file, use the `write_floorplan` command after extraction. This command saves the floorplan information so that you can read the floorplan back into Design Compiler.

Some of the physical constraints imported from the floorplan file are listed below:

- Voltage areas
- Die Area

  The die area represents the silicon boundary of a chip and encloses all objects of a design, such as pads, I/O pins, and cells.
- Placement Area
- Macro Location and Orientation

  For each cell with a location and the FIXED attribute specified, Design Compiler sets the location on the corresponding cell in the design.
- Hard and Soft Placement Blockages

  For defined placement blockages, Design Compiler creates placement blockages on the design.
- Wiring Keepouts

  Wiring keepout information is imported from the floorplan file. The `create_route_guide` command creates a wiring keepout.
- Placement Bounds

  Placement bounds are extracted from the floorplan file in the following two ways:

  1. If there are regions in the design with the same name as in the floorplan file, the cells in the related group are attached to the region by the `update_bounds` command in incremental mode.
  2. If the region does not exist in the design, it is created with the same name as in the floorplan file by applying the `create_bound` command. Matched cells in the related group are also attached.
- Port Locations

For each port with the location specified in the floorplan file, Design Compiler sets the location on the corresponding port in the design.

Ports with changed names and multilayers are supported.

- Preroutes

  Design Compiler imports preroutes that are defined in the floorplan file. A preroute is represented with the `create_net_shape` command.

- Site Array Information

  Design Compiler extracts site array information that is defined in the floorplan file. Site arrays define the placement area.

For command details, see the man page.

Floorplan modification issues and port and macro name matching considerations are described in the following sections:

- Incremental Floorplan Modifications Using the read_floorplan Command
- Matching Names of Macros and Ports

Incremental Floorplan Modifications Using the read_floorplan Command

The `read_floorplan` command should not be used to perform incremental floorplan modifications. The `read_floorplan` command imports the entire floorplan information exported from IC Compiler with the `write_floorplan` command and overwrites any existing floorplan. The `write_floorplan` command usually includes commands to remove any existing floorplan.

Matching Names of Macros and Ports

By default, when the `read_floorplan` command applies physical constraints in topographical mode, it has an intelligent name matching capability that matches macros and ports in the floorplan file with macros and ports in memory. The command uses the intelligent name matching capability when it does not find an exact match.

The `read_floorplan` command reads from floorplan files generated from a netlist that could have different object names from the netlist in memory. These name mismatches can be caused by automatic ungrouping and the `change_names` command. Typically, hierarchy separators and bus notations are sources of these mismatches.

For example, automatic ungrouping by the `compile_ultra` command followed by `change_names` might result in the forward slash (/) separator being replaced with an underscore (_) character. Therefore, a macro named *a/b/c/macro_name* in the RTL might be named *a/b_c_macro_name* in the mapped netlist, which is the input to the back-end tool.

When extracting physical constraints from the DEF file, the `extract_physical_constraints` command automatically resolves these name differences by using Design Compiler's intelligent name matching capability.

By default, the following characters are considered equivalent:

- Hierarchical separators { / _ . }

  For example, a cell named *a.b_c/d_e* is automatically matched with the string *a/b_c.d/e* in the floorplan file.

- Bus notations { [ ] __ ( ) }

  For example, a cell named *a [4] [5]* is automatically matched with the string *a_4__5_* in the floorplan file.

To define the rules used by the intelligent name matching capability, use the `set_fuzzy_query_options` command.

Note:
Using the `source` command is not recommended. However, if you use the `source` command to import your floorplan information, you need to enable fuzzy name matching manually by setting the `fuzzy_matching_enabled` variable to `true` before you source the floorplan file, for example,

```
(set_fuzzy_query_options)
set fuzzy_matching_enabled true
source design.fp
set fuzzy_matching_enabled false
```

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

Manually Defining Physical Constraints

This section describes how to manually define physical constraints in the following subsections:

- Defining Physical Constraints Overview
- Defining the Die Area With the create_die_area Command
- Defining the Core Placement Area With the create_site_rows Command
- Defining Placement Area With the set_aspect_ratio and set_utilization Commands
- Defining Port Locations
- Defining Macro Location and Orientation

- Defining Placement Blockage
- Defining Voltage Area
- Defining Placement Bounds
- Creating Wiring Keepouts
- Creating Preroutes

Defining Physical Constraints Overview

This section describes the commands you can use to define the floorplan's physical constraints manually. DC Ultra, in topographical mode, can read physical constraints (floorplan information) from a Tcl-based script of physical commands.

You manually define physical constraints when you cannot obtain this information from a DEF file or from the `read_floorplan` command. For details, see "Importing Physical Constraints From an IC Compiler DEF Formatted File" on page 10-17 and "Importing Physical Constraints From an IC Compiler write_floorplan Formatted File" on page 10-26. After you have manually defined your physical constraints in a Tcl script file, use the `source` command to apply these constraints. Keep the following points in mind when you manually define physical constraints:

- You must read in the design before applying user-specified physical constraints.
- You must apply user-specified physical constraints during the first topographical mode session.

Table 10-3 lists the commands that you use to set physical constraints.

Table 10-3 Commands That Define Physical Constraints

To define this physical constraint	Use these commands
Die Area (For details, see "Defining the Die Area With the create_die_area Command" on page 10-31.)	`create_die_area`
Floorplan estimate when exact area is not known (For details, see "Defining Placement Area With the set_aspect_ratio and set_utilization Commands" on page 10-33.)	`set_aspect_ratio` and `set_utilization`
Exact core area (For details, see "Defining the Core Placement Area With the create_site_rows Command" on page 10-33.)	`create_site_rows`

Table 10-3 Commands That Define Physical Constraints (Continued)

To define this physical constraint	Use these commands
Relative port locations (See "Defining Relative Port Locations" on page 10-34)	`set_port_side`
Exact port locations (See "Defining Exact Port Locations" on page 10-35)	`set_port_location`
Macro location and orientation (See "Defining Macro Location and Orientation" on page 10-36)	`set_cell_location`
Placement keepout (blockages) (See "Defining Placement Blockage" on page 10-37)	`create_placement_blockage`
Voltage area (See "Defining Voltage Area" on page 10-37)	`create_voltage_area`
Placement bounds (See "Defining Placement Bounds" on page 10-38)	`create_bounds`
Wiring keepouts (See "Creating Wiring Keepouts" on page 10-41)	`create_route_guide`
Preroutes (See "Creating Preroutes" on page 10-41)	`create_net_shape`

Defining the Die Area With the create_die_area Command

DC Ultra topographical technology supports the manual definition of the die area, which is also known as the cell boundary. The die area represents the silicon boundary of a chip, and it encloses all objects of a design, such as pads, I/O pins, and cells.

Typically, you create floorplan constraints, including die area, in your floorplanning tool and import these physical constraints into DC Ultra topographical mode. However, if you are not using a floorplanning tool, you may need to create your physical constraints manually. The `create_die_area` command enables you to define your die area manually from within DC Ultra topographical mode.

When using the `create_die_area` command, you can use the `-coordinate` option if your die area is a rectangle; you must use the `-polygon` option if your die area is rectilinear. Note that you can use the `-polygon` option to specify rectangles, but you cannot use the `-coordinate` option to specify rectilinear die areas. For example, this command

```
create_die_area -coordinate { 0 0 100 100 }
```

or this command

```
create_die_area -coordinate { {0 0} {100 100} }
```

will create the same rectangular die area as the following command:

```
create_die_area -polygon { {0 0} {100 0} {100 100} {0 100} }
```

For the `create_die_area` command syntax details, see the man page.

Defining the Core Placement Area With the create_site_rows Command

Define the core placement area with the `create_site_rows` command.

The core placement area is a box that contains all rows. It represents the placeable area for standard cells. The core area is smaller than the cell boundary (the die area). Pads, I/O pins, and top-level power and ground rings are found outside the core placement area. Standard cells, macros, and wire tracks are typically found inside the core placement area. There should be only one core area in a design.

You use the `create_site_row` command to create a row of sites or a site array at a specified location. A site is a predefined valid location where a leaf cell can be placed; a site array is an array of placement sites and defines the placement core area.

To create a horizontal row of 100 CORE_2H sites with a bottom-left corner located at (10,10) and rows spaced 5 units apart, enter

```
create_site_row -count 100 -kind CORE_2H -space 5 \
                           -coordinate {10 10}
```

Use the `report_area -physical` command to report physical information such as core area, aspect ratio, utilization, total fixed cell area, and total movable cell area.

For the `create_site_rows` command syntax details, see the man page.

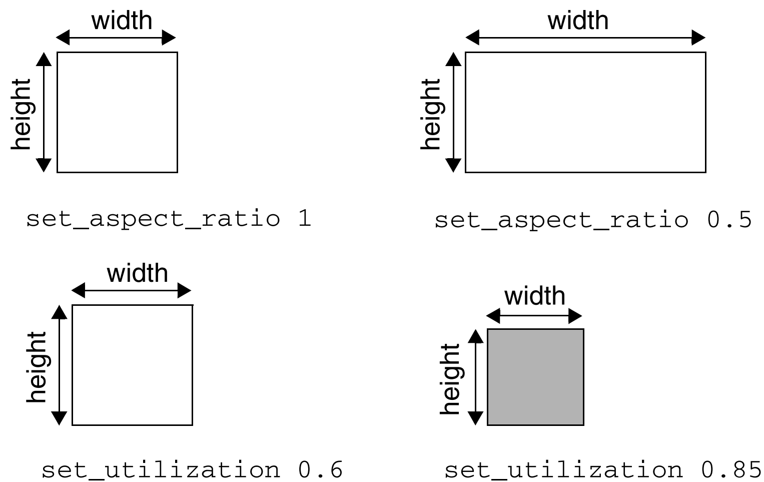
Defining Placement Area With the set_aspect_ratio and set_utilization Commands

If you have not defined your die area with the `create_die_area` command, defined a floorplan area with the `create_site_rows` command, or imported floorplan information from a floorplanning tool, you can use the `set_aspect_ratio` and `set_utilization` commands to estimate the placement area.

The aspect ratio is the height-to-width ratio of a block; it defines the shape of a block. Utilization specifies how densely you want cells to be placed within the block. Increasing utilization reduces the core area.

Figure 10-3 illustrates how to use these commands.

Figure 10-3 Using the set_aspect_ratio and set_utilization Commands



Defining Port Locations

You can define constraints that restrict the placement and sizing of ports by using the `set_port_side` command or `set_port_location` command. You can specify relative or exact constraints.

The following subsections describe how to define relative and exact port locations:

- Defining Relative Port Locations
- Defining Exact Port Locations

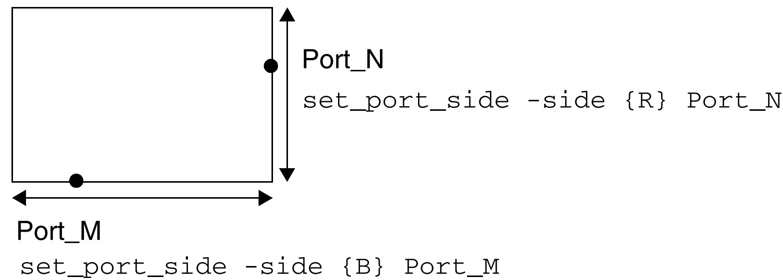
Defining Relative Port Locations

Use the `set_port_side` command to define relative port locations as follows:

```
set_port_side port_name -side {L|R|T|B}
```

Valid sides are left (L), right (R), top (T), or bottom (B). A port can be placed at any location along the specified side. If the port side constraints are provided, the ports are snapped to the specified side. Otherwise, by default, the ports are snapped to the side nearest to the port location assigned by the coarse placer. Figure 10-4 shows how you define port sides by using the `set_port_side` command.

Figure 10-4 Setting Relative Port Sides



Defining Exact Port Locations

Use the `set_port_location` command to annotate the port location on the specified port, defining exact port locations as follows:

```
set_port_location port_name -coordinate {x y}
```

where `-coordinate` is the lower left point of the port shape.

The following example annotates port Z, a top-level port in the current design, with the location at 100, 5000.

```
dc_shell-topo> set_port_location -coord {100 5000} Z
```

In addition, you can use the `-layer_name` option and `-layer_area` option to define metal layer geometry. For example, the following command specifies that the layer geometry is a rectangle of dimensions 10 by 20 for port Z.

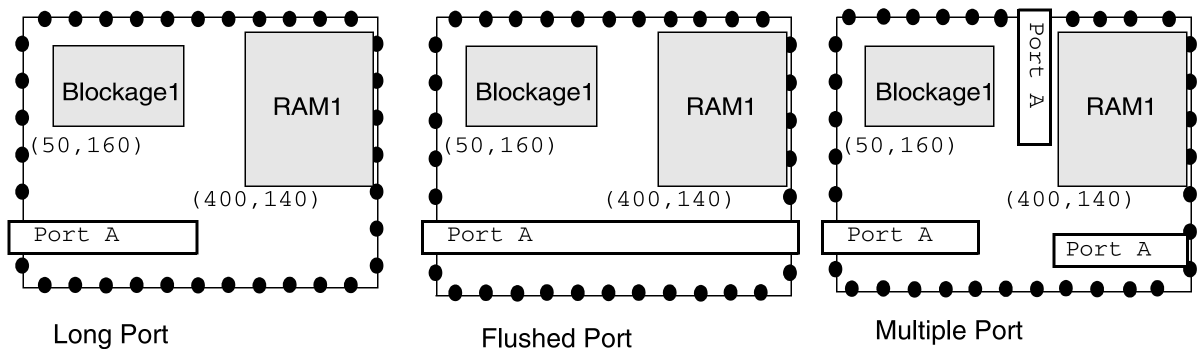
```
dc_shell-topo> set_port_location -layer_name METAL1 \
                     -layer_area {-5 -10 5 10} Z
```

These options allow you to define long ports so that connections to the long port can be made along any point on the specified metal layer during virtual-layout based optimization. A long port can be an input, output, or bidirectional port. You specify a long port location by using a dimension greater than the minimum metal area. Figure 10-5 illustrates the different types of port dimensions: Long port, flushed port, and multiple port.

- A long port has the metal area defined with a single metal layer and touches only one side of the block core area boundary.
- A flushed port has the metal area defined with a single metal layer and touches the block core area boundary on two sides.

- A multiple port has more than one connection to the block and can touch the block boundary on two or more sides and can use one or more layers.To create a multiple port, use the `-append` option of the `set_port_locations` command. With this option you add multiple shapes to a port and create a multiple port.

Figure 10-5 Types of Port Dimension Specifications



Defining Macro Location and Orientation

You can define exact macro locations by setting the following command:

```
set_cell_location cell_name -coordinate {x y} -fixed
                            -orientation {N|S|E|W|FN|FS|FE|FW}
```

where `-coordinate` is the coordinate of the lower left corner of the cell's bounding box. The coordinate numbers are displayed in microns relative to the block orientation. The orientation value is one of the rotations listed in the following table.

Orientation	Rotation
N (default)	Nominal orientation, 0 degree rotation (north)
S	180 degrees (south)
E	270 degrees counterclockwise (east)
W	90 degrees counterclockwise (west)
FN	Reflection in the y-axis (flipped north)
FS	180 degrees, followed by reflection in the y-axis (flipped south)
FE	270 degrees counterclockwise, followed by reflection in the y-axis (flipped east)
FW	90 degrees counterclockwise, followed by reflection in the y-axis (flipped west)

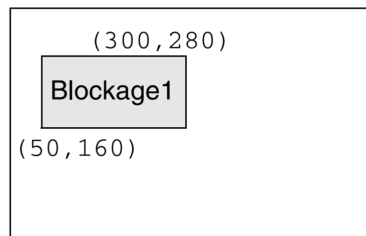
Defining Placement Blockage

Use the `create_placement_blockage` command to define hard, soft, and partial placement blockages.

Figure 10-6 uses the `create_placement_blockage` command to define a hard placement blockage named Blockage1.

Figure 10-6 Defining a Hard Placement Blockage

```
create_placement_blockage\
-name Blockage1 -type hard \
-coordinate {50 160 300 280}
```



Defining Voltage Area

You define voltage areas by using the `create_voltage_area` command. This command enables you to create a voltage area on the core area of the chip. The voltage area is associated with hierarchical cells. The tool assumes the voltage area to be an exclusive, hard move bound and tries to place all the cells associated with the voltage area within the defined voltage area, as well as place all the cells not associated with the voltage area outside the defined voltage area.

Example 10-27 uses the `create_voltage_area` command to constrain the instance INST_1 to lie within the voltage area whose coordinates are lower-left corner (100 100) upper-right corner (200 200).

Example 10-27 Using the create_voltage_area Command With the -names Option

```
dc-shell-topo> create_voltage_area -name foo \
   -coordinate {100 100 200 200} INST_1
```

Example 10-28 uses the `create_voltage_area` command to constrain cells in power domain PD1 to lie within the voltage area whose coordinates are lower-left corner (100 100) upper-right corner (200 200).

Example 10-28 Using the create_voltage_area Command With the -power_domain Option

```
dc-shell-topo> create_voltage_area -power_domain PD1 \
   -coordinate {100 100 200 200}
```

Note:

Note: The `-names` option and the `-power_domain` option of the `create_voltage_area` command are mutually exclusive.

In addition to the `create_voltage_area` command, the following commands are supported: `remove_voltage_area` and `report_voltage_area`

Voltage areas are automatically defined when you import your floorplan information from IC Compiler with the `read_floorplan` command. For details, see "Importing Physical Constraints From an IC Compiler write_floorplan Formatted File" on page 10-26. You need to define voltage areas manually when importing floorplan information from a DEF file.

To visually inspect your defined voltage areas, use the layout view in the Design Vision layout window.

Defining Placement Bounds

This section describes how to create and use placement bounds in the following subsections:

- Placement Bounds Overview
- Creating Placement Bounds
- Using Placement Bounds Effectively
- Order for Creating Placement Bounds
- Guidelines for Defining Placement Bounds Effectively
- Summary of Bounds Commands

Placement Bounds Overview

A placement bound is a rectangular or rectilinear area within which to place cells and hierarchical cells. You use the `create_bounds` command to specify placement constraints for coarse placement.

Defining a placement bound enables you to group cells such as clock-gating cells or extremely timing-critical groups of cells that you want to guarantee will not be disrupted for placement by other logic. During placement, the tool ensures that the cells you grouped remain together. Placement bounds are placement constraints.

Creating Placement Bounds

Use the `create_bounds` command to specify placement bounds. You can specify two different types of bounds: move bounds and group bounds. Move bounds restrict the placement of cells to a specific region of the core area. Move bounds require absolute coordinates to be specified, using the `-coordinate` option. Group bounds, on the other hand, are floating region constraints, using the `-dimension` option. Cells in the same group bound are placed within a specified bound but the absolute coordinates are not fixed. Instead, they are optimized by the placer. If you do not use either the `-dimension` or `-coordinate` option, the tool creates a group bound with the bounding box computed internally by the tool.

In addition, move bounds can be soft, hard, or exclusive. Group bounds can be soft or hard.

- Soft bounds specify placement goals, with no guarantee that the cells will be placed inside the bounds. If timing or congestion cost is too high, cells might be placed outside the region. This is the default.
- Hard bounds force placement of the specified cells inside the bounds. To specify hard bounds, use the `-type hard` option along with the `-dimension` or `-coordinate` option. However, overusing hard bounds can lead to inferior placement solutions.
- Exclusive bounds force the placement of the specified cell inside the bounds. All other cells must be placed outside the bounds. To specify exclusive bounds, use the `-exclusive` option.

You can specify the following types of placement bounds:

- Soft group bound

```
create_bounds -dimension {100 100} -name foo1 INST1
```

- Soft move bound

```
create_bounds -coordinate {0 0 10 10} -name foo2 INST2
```

- Hard group bound

```
create_bounds -dimension {100 100} -type hard \
                     -name foo3 INST3
```

- Hard move bound

```
create_bounds -coordinate {0 0 10 10} -type hard \
                     -name foo4 INST4
```

- Exclusive move bound

```
create_bounds -coordinate {0 0 10 10} -type hard \
                     -name foo5 INST5
```

Using Placement Bounds Effectively

Usually, you do not impose bounds constraints, allowing the tool full flexibility to optimize placement for timing and routability. However, if QoR does not meet your requirements, you might improve QoR by using the `create_bounds` command.

Make the number of cells you place in placement bounds relatively small compared with the total number of cells in the design. When you define placement bounds, the solution space available to the placer to reach the optimal result gets smaller.

Order for Creating Placement Bounds

If you impose bounds constraints, create bounds in the following order:

1. Floating group bounds where location and dimension are optimized by the tool. For these bounds, do not specify dimensions.
2. Floating group bounds with fixed dimensions. For these bounds, specify dimensions.
3. Fixed move bounds with fixed location and dimension. For these bounds, specify coordinates that define the bound.

Guidelines for Defining Placement Bounds Effectively

Use the following guidelines when you create placement bounds:

- Use soft and hard move bounds sparingly in your design.
- Avoid placing cells in more than one bound.
- Be aware that including small numbers of fixed cells in group bounds can move the bound.
- Do not use placement bounds as keepouts.
- Maintain even cell density distribution over the chip.

Summary of Bounds Commands

Table 10-4 summarizes the commands related to placement bounds.

Table 10-4 Summary of Bounds Commands

Command	Description
`create_bounds`	Creates rectangular and rectilinear move bounds and group bounds
`update_bounds`	Updates a bound by adding or removing contents
`remove_bounds`	Removes bounds set using `create_bounds`
`report_bounds`	Reports bounds and bound IDs set using `create_bounds`

Creating Wiring Keepouts

You can create wiring keepouts by using the `create_route_guide` command. Example 10-29 uses the `create_route_guide` command to create a keepout named my_keepout_1 in the METAL1 layer at coordinates {12 12 100 100}.

Example 10-29

```
dc_shell-topo> create_route_guide -name  "my_keepout_1"  \
                                -no_signal_layers "METAL1" \
                                -coord {12 12 100 100}
```

Creating Preroutes

You can preroute a group of nets, such as clock nets, before routing the rest of the nets in the design. During global routing, the tool considers these preroutes while computing the congestion map; this map is consistent with IC Compiler. This feature also addresses correlation issues caused by inconsistent floorplan information.

To define preroutes, you use the `create_net_shape` command. You can create three different types of net shapes as shown in Figure 10-7: path, wire (horizontal and vertical), and rectangle.

Figure 10-7 Types of Preroutes

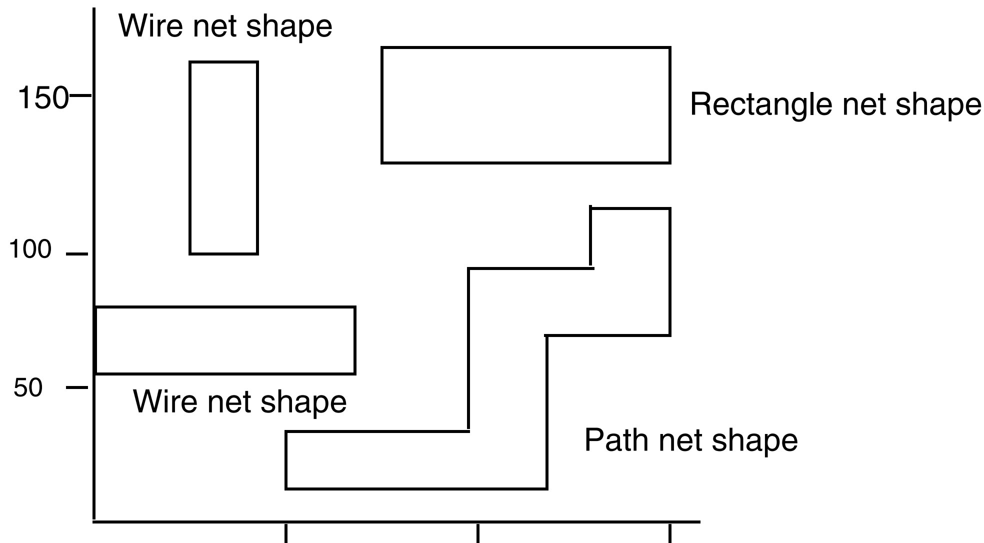


Table 10-5 shows how you use some of the options of the `create_net_shape` command.

Table 10-5 Using the create_net_shape Command Options to Create Preroutes

To do this	Use this option
Specify the type of net shape.	`-type path | wire | rect`
Specify the type of net.	`-net_type ground | power | clock | signal`
Specify the net on which the net shape is created, for example VSS or VDD.	`-net name`
Specify the layer on which the net shape is created.	`-layer name`
Specify the origin of the net shape for wires.	`-origin x y`
Specify the bounding box of the net shape for rectangles.	`-bbox llx lly urx ury`
Specify the point sequence of the net shape for paths.	`-points {x0 y0 x1 y1... xn yn}`
Specify the alignment type of a wire or path. Default is square.	`-path_type { square | round | extend_half_width | octagon }`

Table 10-5 Using the create_net_shape Command Options to Create Preroutes (Continued)

To do this	Use this option
Specify the length and width for wires.	`-length real_number -width real_number`
Specify the type of route.	`-route_type {user_enter | signal_route | signal_route_global | signal_route_detail | pg_ring | pg_strap | pg_macro_io_pin_conn | pg_std_cell_pin_conn | clk_ring | clk_strap | clk_zero_skew_route | bus | shield | shield_dynamic | clk_fill_track}`
Specify the vertical orientation (Default is horizontal).	`-vertical`

The following examples show how to create the net shapes shown in Figure 10-7.

Example 10-30 uses the `create_net_shape` command to create two wire net shapes.

Example 10-30
```
create_net_shape -type wire -net VSS \
      -bbox {0 60 80 80} \
      -layer METAL5 \
      -route_type pg_strap \
      -net_type ground
create_net_shape -type wire -net VSS \
      -origin {50 100} \
      -width 20 \
      -length 80 \
      -layer METAL4 -route_type pg_strap -vertical
```

Example 10-31 uses the `create_net_shape` command to create the path net shape.

Example 10-31
```
create_net_shape -type path -net VSS\
      -points {70 30 110 30 110 90 150 90 150 110} \
      -width 0.20 -layer M3 \
      -route_type pg_std_cell_pin_conn
```

Example 10-32 uses the `create_net_shape` command to create the rectangle net shape.

Example 10-32
```
create_net_shape -type rect -net VSS \
      -bbox {{80 140} {160 180}} \
      -layer METAL1 -route_type pg_strap
```

For syntax and additional details, see the man page.

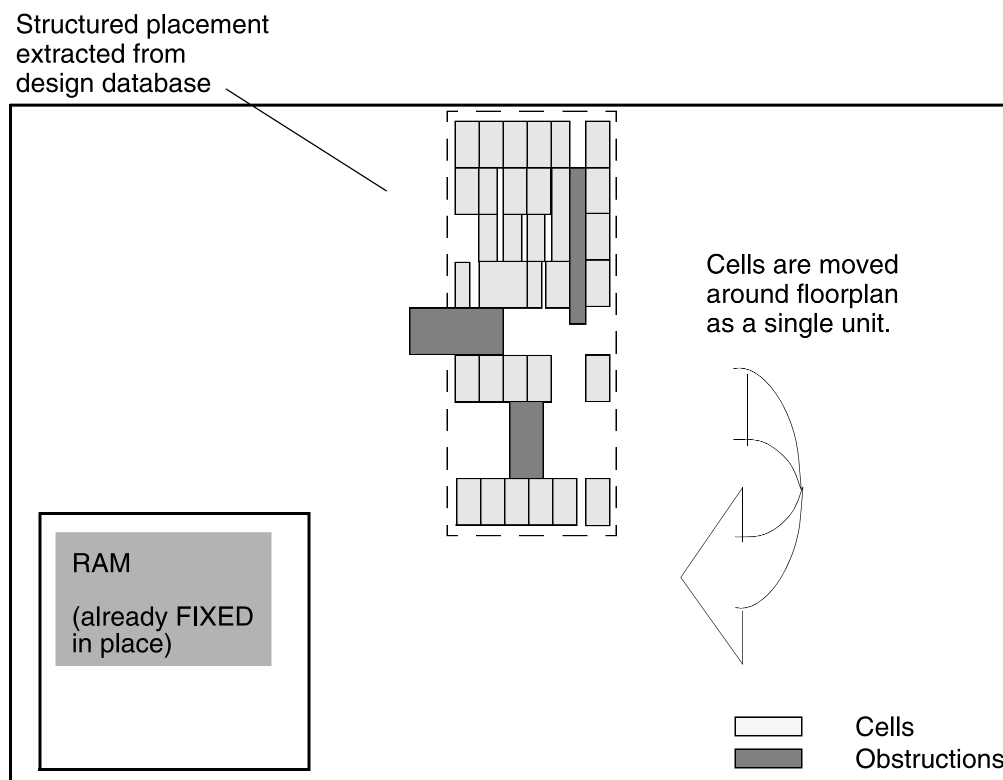
Specifying Relative Placement

The relative placement capability provides a way for you to create structures in which you specify the relative column and row positions of instances. During placement and optimization, these structures are preserved and the cells in each structure are placed as a single entity.

Relative placement is usually applied to datapaths and registers, but you can apply it to any cells in your design, controlling the exact relative placement topology of gate-level logic groups and defining the circuit layout. You can use relative placement to explore QoR benefits, such as shorter wire lengths, reduced congestion, better timing, skew control, fewer vias, better yield, and lower dynamic and leakage power.

The relative placement constraints implicitly generate a matrix structure of instances and control the placement of these instances. You use the resulting annotated netlist for optimization, during which the tool preserves the structure and places it as a single entity or group, as shown in Figure 10-8.

Figure 10-8 Relative Placement in a Floorplan



Relative placement is described in the following subsections:

- Benefits of Relative Placement
- Methodology for the Relative Placement Flow
- Summary of Relative Placement Commands
- Creating Relative Placement Using Compiler Directives
- Creating Relative Placement Groups
- Anchoring Relative Placement Groups
- Applying Compression to Relative Placement Groups
- Specifying Alignment
- Adding Objects to a Group
- Querying Relative Placement Groups
- Checking Relative Placement Constraints
- Saving Relative Placement Information
- Removing Relative Placement Group Attributes
- Sample Script for a Relative Placement Flow

Benefits of Relative Placement

Relative placement provides the following benefits:

- Provides a method for maintaining structured placement for legacy or intellectual property (IP) designs
- Reduces the placement search space in critical areas of the design, which means greater predictability of QoR (wire length, timing, power)
- Correlates better with IC Compiler
- Can minimize congestion and improve routability

Methodology for the Relative Placement Flow

The methodology for the relative placement flow follows these major steps:

1. Read a mapped gate-level netlist into topographical mode.
2. Define the relative placement constraints.
   a. Create the relative placement groups by using the `create_rp_group` command. See “Creating Relative Placement Groups” on page 10-49.
   b. Add relative placement objects to the groups by using the `add_to_rp_group` command. See “Adding Relative Placement Groups” on page 10-59.

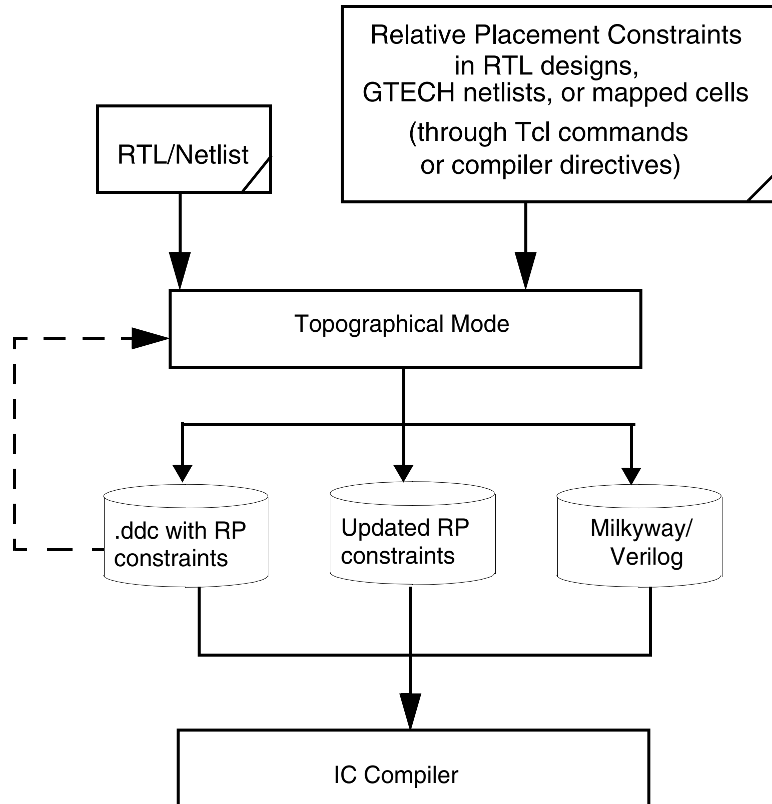
   Topographical mode annotates the netlist with the relative placement constraints and places an implicit `size_only` constraint on these cells.
3. Read floorplan information. For example, enter

   ```
   dc_shell> extract_physical_constraints floorplan.def
   ```
4. Check the relative placement by using the `check_rp_groups` command.

   The command reports relative placement failures, such as:

   ```
   RPGP-028: The height ‘%f’ of the RP group ‘%s’ is more than the height
   ‘%f’ of the core area.
   RPGP-035 (warning) Relative placement leaf cell data may have been
   lost.
   RPGP-0348: All RP cells in cluster ‘%s’ are not within same voltage
   region;
   ignoring voltage region placement.
   ```
5. Synthesize and optimize the design by using the `compile_ultra` command. Figure 10-9 shows the relative placement flow in topographical mode.
6. To visually verify placement, use the layout view in the Design Vision layout window.

Figure 10-9 Relative Placement Flow



Keep the following points in mind when you use the relative placement feature:

- Topographical mode automatically places a `size_only` attribute on the relative placement cells to preserve the RP structure.
- Relative placement constraints are handled appropriately and preserved by using the `uniquify` and `ungroup` commands.
- New cells added during optimization between two relative placement cells do not automatically belong to the relative placement group.
- Make sure that relative placement is applicable to your design. A design can contain both structured and unstructured elements. Some designs such as datapaths and pipelined designs are more appropriate for structured placement. Providing relative placement constraints for cells that would have be placed better by the tool can deliver poor results.
- Relative placement constraints are carried in the .ddc file for subsequent topographical mode sessions.

Summary of Relative Placement Commands

You can specify relative placement constraints by using a dedicated set of Tcl commands, similar to the commands in IC Compiler. Table 10-6 summarizes the commands available in Design Compiler topographical mode for relative placement. The sections that follow describe how to use these commands.

Table 10-6 Summary of Relative Placement Commands

Command	Description
`create_rp_group`	Creates new relative placement groups.
`add_to_rp_group`	Adds items to relative placement groups.
`set_rp_group_options`	Sets relative placement group attributes.
`report_rp_group_options`	Reports attributes for relative placement groups.
`get_rp_groups`	Creates a collection of relative placement groups that match certain criteria.
`write_rp_groups`	Writes out relative placement information for specified groups.
`all_rp_groups`	Returns a collection of specified relative placement groups and all subgroups in their hierarchy.
`all_rp_hierarchicals`	Returns a collection of hierarchical relative placement groups that are ancestors of specified groups.
`all_rp_inclusions`	Returns a collection of hierarchical relative placement groups that include specified groups.
`all_rp_instantiations`	Returns a collection of hierarchical relative placement groups that instantiate specified groups.
`all_rp_references`	Returns a collection of relative placement groups that contain specified cells (either leaf cells or hierarchical cells that contain instantiated relative placement groups).
`check_rp_groups`	Checks relative placement constraints and reports failures.
`remove_rp_groups`	Removes a list of relative placement groups.
`remove_rp_group_options`	Reports attributes for the specified relative placement groups.

Table 10-6 Summary of Relative Placement Commands (Continued)

Command	Description
remove_from_rp_group	Removes an item (cell, relative placement group, or keepout) from the specified relative placement groups.
rp_group_inclusions	Returns collections for directly embedded included groups (added to a group by using the add_rp_group -hierarchy command) in all or specified groups.
rp_group_instantiations	Returns collections for directly embedded instantiated groups (added to a group by using the add_rp_group -hierarchy -instance command) in all or specified groups.
rp_group_references	Returns collections for directly embedded leaf cells (added to a group by using the add_rp_group -leaf command), directly embedded included cells that contain hierarchically instantiated cells (added to the included group by using the add_rp_group -hierarchy -instance command), or both in all or specified relative placement groups.

Creating Relative Placement Using Compiler Directives

DC Ultra topographical mode supports relative placement information embedded within the Verilog or VHDL description. This capability is enabled by compiler directives that can specify and modify relative placement information. Using these embedded compiler directives, relative placement constraints can be placed in an RTL design, a GTECH netlist, or a mapped netlist. Using compiler directives to specify relative placement increases design flexibility and simplifies relative placement because you no longer need to update the location of many of the cells in the design.

For information about specifying relative placement data for RTL designs, GTECH netlists, and mapped netlists, see the "Creating Relative Placement in Hardware Description Languages," section in the *HDL Compiler for VHDL User Guide* and the *HDL Compiler for Verilog User Guide*.

Creating Relative Placement Groups

A relative placement group is an association of cells, other groups, and keepouts. A group is defined by the number of rows and columns it uses. To create a relative placement group in Design Compiler topographical mode, use the create_rp_group command. Topographical mode creates a relative placement group named *design_name*::*group_name*, where *design_name* is the design specified by the -design option or the current design if you do not use the -design option. You must use this name or a collection of relative placement groups when referring to this group in other relative placement commands. See the man page for a list of available options.

If you do not specify any options, the tool creates a relative placement group that has one column and one row. The group will not contain any objects. To add objects (leaf cells, relative placement groups, or keepouts) to a relative placement group, use the `add_to_rp_group` command, which is described in "Adding Objects to a Group" on page 10-54.

For example, to create a group named designA::rp1, having six columns and six rows, enter

```
create_rp_group rp1 -design designA -columns 6-rows 6
```

Figure 10-10 shows the positions of columns and rows in a relative placement group.

Figure 10-10 Relative Placement Column and Row Positions

In this figure,

- Columns count from column 0 (the leftmost column).
- Rows count from row 0 (the bottom row).
- The width of a column is the width of the widest cell in that column.
- The height of a row is determined by the height of the tallest cell in that row.
- It is not necessary to use all positions in the structure. For example, in this figure, positions 0 3 (column 0, row 3) and 4 1 (column 4, row 1) are not used.

Table 10-7 describes some options you can use with the create_rp_group command.

Table 10-7 Using the create_rp_group Command Options

To do this	Use this option
Specify the anchor location.	-x_offset -y_offset
Specify the type of alignment used by the group.	-alignment
Specify the group alignment pin.	-pin_align_name
Specify the utilization percentage (default is 100 percent).	-utilization
Ignore this relative placement group.	-ignore
Apply compression in the horizontal direction.	-compress

Anchoring Relative Placement Groups

By default, topographical mode can place a relative placement group anywhere within the core area. You can control the placement of a top-level relative placement group by anchoring it.

To anchor a relative placement group, use the `create_rp_group` or the `set_rp_group_options` command with the `-x_offset` and `-y_offset` options. The offset values are float values, in microns, relative to the chip's origin.

If you specify both the x- and y-coordinates, the group is anchored at that location. If you specify only one coordinate, IC Compiler can determine the placement by sliding the group along the unspecified coordinate.

For example, to specify a relative placement group anchored at (100, 100), enter the following command:

```
create_rp_group misc1 -design block1 \
          -columns 3 -rows 10 -x_offset 100 -y_offset 100
```

Applying Compression to Relative Placement Groups

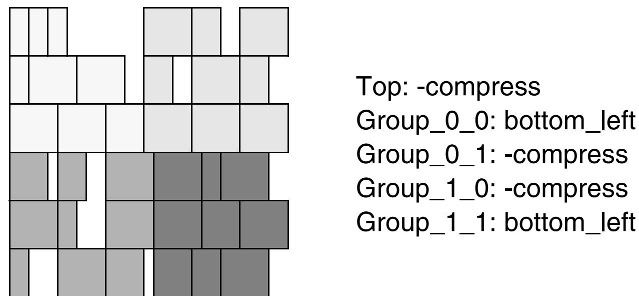
You can apply compression to a relative placement group in the horizontal direction during placement by using the `-compress` option with the `create_rp_group` command or the `set_rp_group_options` command. Setting this option enables bit-stack placement that places each row of a relative placement group without any gaps between leaf cells, lower-level hierarchical relative placement groups, or keepouts. Note that column alignment is not maintained when you use compression.

If you specify both the -utilization and -compress options, the utilization constraints are observed with gaps between leaf elements in a relative placement row. The -compress option does not propagate from a parent group to child groups. To disable relative placement with compression, use the remove_rp_group_options -compress command.

Supporting Compression with Mixed Alignment

Relative placement groups with alignment, such as bottom left, bottom right, or pin alignment, and the relative placement group that is created by using -compress can be placed on the same top-level groups as shown in Figure 10-11. The individual groups of the top level are aligned with compression only if you specify the -compress option. Note that the compression specified at the top level does not propagate to the child groups. The default alignment of the top-level groups is bottom left and the -compress option is disabled by default.

Figure 10-11 Compression of Relative Placement Groups with Mixed Alignment



Specifying Alignment

To specify the default alignment method to use when placing leaf cells and relative placement groups, use the -alignment option with the create_rp_group or set_rp_group_options command.

```
set_rp_group_options -alignment bottom-right [get_rp_groups *]
```

Controlling the cell alignment can improve the timing and routability of your design. You can specify a bottom-left, bottom-right or bottom-pin alignment. If you do not specify an option, the tool uses a bottom-left alignment.

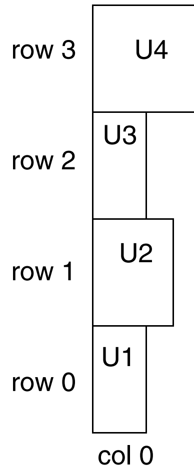
The script in Example 10-33 defines a relative placement group that is bottom-left aligned. The resulting structure is shown in Figure 10-12.

Example 10-33 Definition for Bottom-Left-Aligned Relative Placement Group

```
create_rp_group rp1 -design pair_design -columns 1 -rows 4
   add_to_rp_group pair_design::rp1 -leaf U1 -column 0 -row 0
   add_to_rp_group pair_design::rp1 -leaf U2 -column 0 -row 1
```

```
add_to_rp_group pair_design::rp1 -leaf U3 -column 0 -row 2
add_to_rp_group pair_design::rp1 -leaf U4 -column 0 -row 3
```

Figure 10-12 Bottom-Left-Aligned Relative Placement Group



To align a group by pin location, use the `-alignment bottom-pin` and `-pin_align_name` options of the `create_rp_group` or `set_rp_group_options` command.

```
set_rp_group_options -alignment bottom-pin
   -pin_align_name align_pin
```

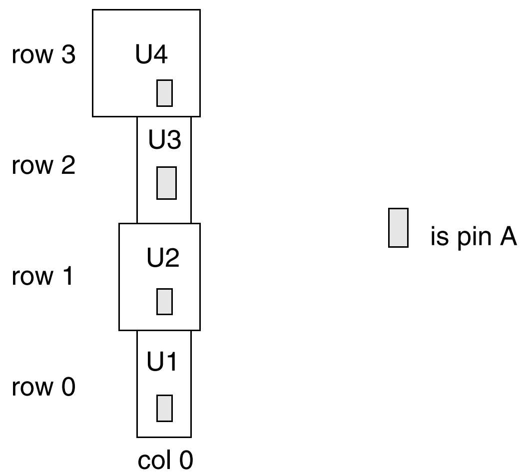
Topographical mode looks for the specified alignment pin in each cell in the column. If the alignment pin exists in a cell, the cell is aligned by use of the pin location. If the specified alignment pin does not exist in a cell, the cell is aligned at the bottom-left corner and the tool generates an information message. If the specified alignment pin does not exist in any cell in the column, the tool generates a warning message.

The script in Example 10-34 defines a relative placement group that is aligned by pin A. The resulting structure is shown in Figure 10-13.

Example 10-34 Definition for Relative Placement Group Aligned by Pins

```
create_rp_group rp1 -design pair_design -columns 1 -rows 4 -pin_align_name A
   add_to_rp_group pair_design::rp1 -leaf U1 -column 0 -row 0
   add_to_rp_group pair_design::rp1 -leaf U2 -column 0 -row 1
   add_to_rp_group pair_design::rp1 -leaf U3 -column 0 -row 2
   add_to_rp_group pair_design::rp1 -leaf U4 -column 0 -row 3
```

Figure 10-13 Relative Placement Group Aligned by Pins



When you specify an alignment pin for a group, the pin applies to all cells in the group. You can override the group alignment pin for specific cells in the group by specifying the `-pin_align_name` option when you use the `add_to_rp_group` command to add the cells to the group.

Adding Objects to a Group

You can add leaf cells, other relative placement groups, and keepouts to relative placement groups (created with the `create_rp_group` command). You use the `add_to_rp_group` command to add objects.

When you add an object to a relative placement group, keep the following points in mind:

- The relative placement group to which you are adding the object must exist.
- The object must be added to an empty location in the relative placement group.

Adding Leaf Cells

To add a leaf cell to a relative placement group, use the `add_to_rp_group` command. In a relative placement group, a leaf cell can occupy multiple column positions or multiple row positions, which is known as leaf cell straddling. You can create a more compact relative placement group by straddling leaf cells. To define straddling, you specify multiple column or row positions by using the `-num_columns` or `-num_rows` options respectively. If you do not specify these options, the default is 1. For example, to create a leaf cell of two columns and one row, enter

```
add_to_rp_group rp_group_name -leaf cell_name \
-column 0 -num_columns 2 -row 0 -num_rows 1
```

You should not place a relative placement keepout at the same location of a straddling leaf cell. In addition, straddling is for leaf cells only, but not for hierarchical groups or keepouts.

Note:
You should not apply compression to a straddling leaf cell that has either multiple column positions, multiple row positions, or both. You can apply right alignment or pin alignment to a straddling leaf cell with multiple row positions, but not to a cell with multiple column positions.

Include the `-orientation` option with a list of possible orientations when you add the cells to the group with the `add_to_rp_group` command. See the man page for a list of available options.

Aligning Leaf Cells Within a Column

You can align the leaf cells in a column of a relative placement group by using the following alignment methods:

- Bottom left (default)
- Bottom right
- Pin alignment

Controlling the cell alignment can improve the timing and routability of your design.

Aligning by Bottom-Left Corners

To align the leaf cells by aligning the bottom-left corners, use the `-alignment bottom-left` option with the `create_rp_group` command or the `set_rp_group_options` command:

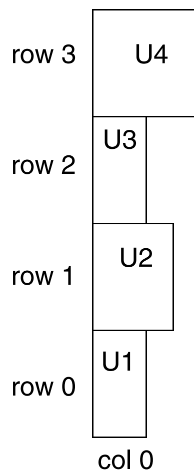
`set_rp_group_options -alignment bottom-left [get_rp_groups *]`

The script in Example 10-35 defines a relative placement group that is bottom-left aligned. The resulting structure is shown in Figure 10-14.

Example 10-35 Definition for Bottom-Left Aligned Relative Placement Group

```
create_rp_group rp1 -design pair_design -columns 1 -rows 4
   add_to_rp_group pair_design::rp1 -leaf U1 -column 0 -row 0
   add_to_rp_group pair_design::rp1 -leaf U2 -column 0 -row 1
   add_to_rp_group pair_design::rp1 -leaf U3 -column 0 -row 2
   add_to_rp_group pair_design::rp1 -leaf U4 -column 0 -row 3
```

Figure 10-14 Bottom-Left-Aligned Relative Placement Group



Aligning by Bottom-Right Corners

To align a group by aligning the bottom-right corners, use the `-alignment bottom-right` option with the `create_rp_group` command or the `set_rp_group_options` command:

```
set_rp_group_options -alignment bottom-right \
   [get_rp_groups *]
```

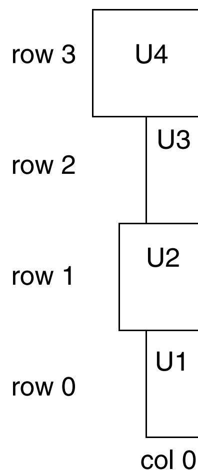
Note:
For hierarchical relative placement groups, the bottom-right alignment does not propagate through the hierarchy.

The script in Example 10-36 defines a relative placement group that is bottom-right aligned. The resulting structure is shown in Figure 10-15.

Example 10-36 Definition for Bottom-Right Aligned Relative Placement Group

```
create_rp_group rp1 -design pair_design -columns 1 -rows 4 \
   -alignment bottom-right
   add_to_rp_group pair_design::rp1 -leaf U1 -column 0 -row 0
   add_to_rp_group pair_design::rp1 -leaf U2 -column 0 -row 1
   add_to_rp_group pair_design::rp1 -leaf U3 -column 0 -row 2
   add_to_rp_group pair_design::rp1 -leaf U4 -column 0 -row 3
```

Figure 10-15 Bottom-Right Aligned Relative Placement Group



Aligning by Pin Location

To align a group by pin location, use the `-alignment bottom-pin` and `-pin_align_name` options of the `create_rp_group` or `set_rp_group_options` command.

```
set_rp_group_options -alignment bottom-pin \
   -pin_align_name align_pin
```

Design Compiler looks for the specified alignment pin in each cell in the column. If the alignment pin exists in a cell, the cell is aligned by use of the pin location. If the specified alignment pin does not exist in a cell, the cell is aligned at the bottom-left corner and Design Compiler generates an information message. If the specified alignment pin does not exist in any cell in the column, Design Compiler generates a warning message.

If you specify both pin alignment and cell orientation, Design Compiler resolves potential conflicts as follows:

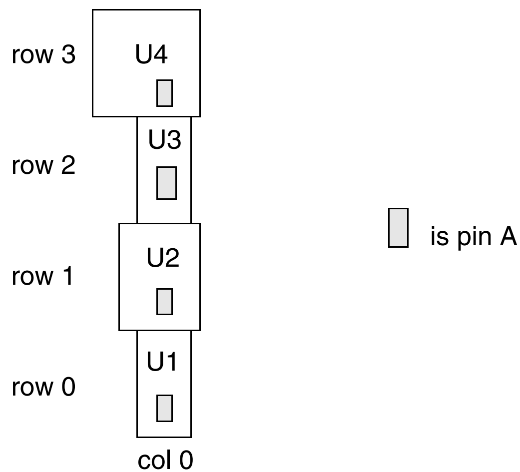
- User specifications for cell orientation take precedence over the pin alignment done by Design Compiler.
- Pin alignment done by Design Compiler takes precedence over the cell orientation optimization done by Design Compiler.

The script in Example 10-37 defines a relative placement group that is aligned by pin A. The resulting structure is shown in Figure 10-16.

Example 10-37 Definition for Relative Placement Group Aligned by Pins

```
create_rp_group rp1 -design pair_design -columns 1 -rows 4 -pin_align_name A
   add_to_rp_group pair_design::rp1 -leaf U1 -column 0 -row 0
   add_to_rp_group pair_design::rp1 -leaf U2 -column 0 -row 1
   add_to_rp_group pair_design::rp1 -leaf U3 -column 0 -row 2
   add_to_rp_group pair_design::rp1 -leaf U4 -column 0 -row 3
```

Figure 10-16 Relative Placement Group Aligned by Pins



When you specify an alignment pin for a group, the pin applies to all cells in the group. You can override the group alignment pin for specific cells in the group by specifying the `-pin_align_name` option when you use the `add_to_rp_group` command to add the cells to the group.

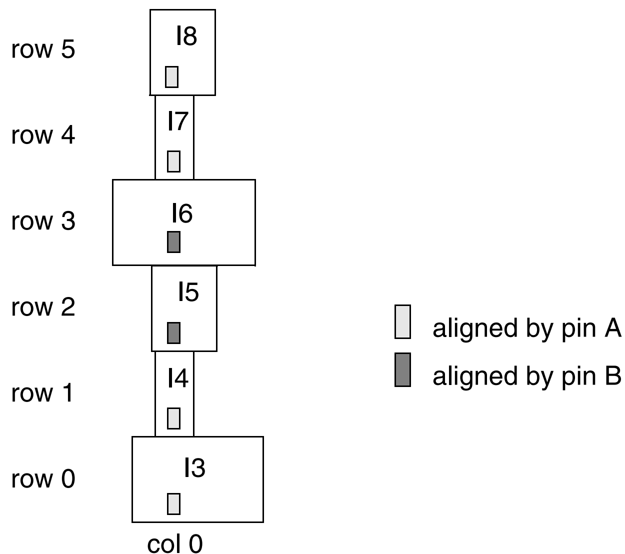
Note:
You cannot specify a cell-specific alignment pin when you add a leaf cell from the relative placement hierarchy browser.

The script in Example 10-38 defines relative placement group misc1, which uses pin A as the group alignment pin; however, instances I5 and I6 use pin B as their alignment pin, rather than the group alignment pin. The resulting structure is shown in Figure 10-17.

Example 10-38 Definition for Aligning a Group and Leaf Cells by Pins

```
create_rp_group misc1 -design block1 -columns 3 -rows 10 \
   -pin_align_name A
   add_to_rp_group block1::misc1 -leaf I3 -column 0 -row 0
   add_to_rp_group block1::misc1 -leaf I4 -column 0 -row 1
   add_to_rp_group block1::misc1 -leaf I5 -column 0 -row 2 \
      -pin_align_name B
   add_to_rp_group block1::misc1 -leaf I6 -column 0 -row 3 \
      -pin_align_name B
   add_to_rp_group block1::misc1 -leaf I7 -column 0 -row 4
   add_to_rp_group block1::misc1 -leaf I8 -column 0 -row 5
```

Figure 10-17 Relative Placement Group Aligned by Pins



Adding Relative Placement Groups

Hierarchical relative placement allows relative placement groups to be embedded within other relative placement groups. The embedded groups then are handled similarly to leaf cells. You can use hierarchical relative placement to simplify the expression of relative placement constraints. With hierarchical relative placement, you do not need to provide relative placement information multiple times for a recurring pattern.

There are two methods for adding a relative placement group to a hierarchical group. You can include the group or instantiate the group. You use the `add_to_rp_group` command for both methods:

- Include the group

  If the relative placement group to be added is in the same design as its parent group, it is an included group. You can include groups in either flat or hierarchical designs. When you include a relative placement group in a hierarchical group, it is as if the included group is directly embedded within its parent group. An included group can be used only in a group of the same design and only once. However, a group that contains an included group can be further included in another group in the same design or can be instantiated in a group of a different design.

- Instantiate the group

  If the relative placement group to be added is in an instance of a subdesign of its parent group, it is an instantiated group. You can instantiate groups only in hierarchical designs.

The group specified in the `-hierarchy` option must be defined in the reference design of the instance specified in the `-instance` option. In addition, the specified instance must be in the same design as the hierarchical group in which you are instantiating the specified group. Using an instantiated group is a useful way to replicate relative placement information across multiple instances of a design and to create relative placement relationships between those instances.

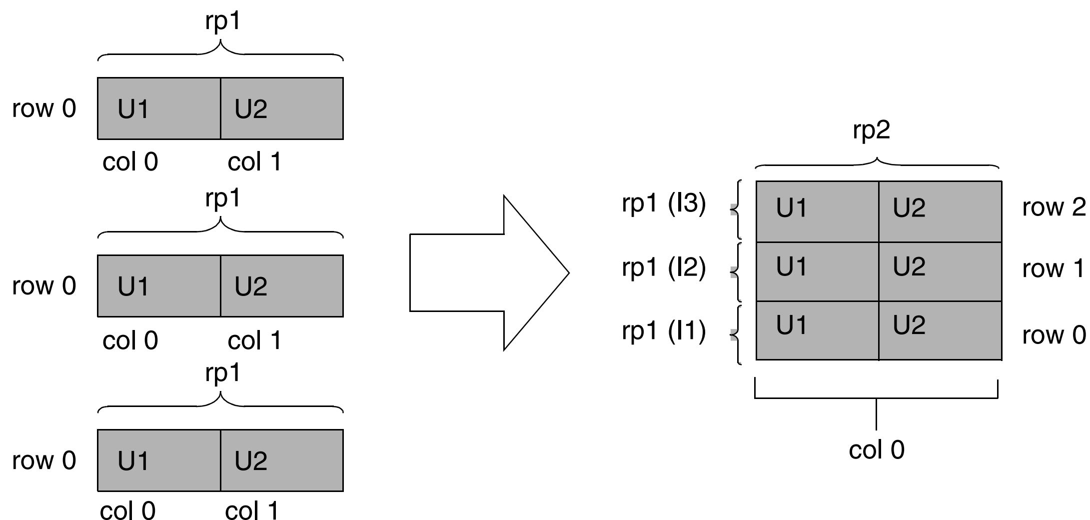
The script in Example 10-39 creates a hierarchical group (rp2) that contains three instances of group rp1. Group rp1 is in the design pair_design and includes leaf cells U1 and U2. Group rp2 is a hierarchical group in the design mid_design that instantiates group rp1 three times (mid_design must contain at least three instances of pair_design). Group rp2 is treated as a leaf cell. You can instantiate rp2 multiple times and in multiple places, up to the number of times mid_design is instantiated in your netlist.

The resulting hierarchical relative placement group is shown in Figure 10-18.

Example 10-39 Instantiating Groups in a Hierarchical Group

```
create_rp_group rp1 -design pair_design -columns 2 -rows 1
   add_to_rp_group pair_design::rp1 -leaf U1 -column 0 -row 0
   add_to_rp_group pair_design::rp1 -leaf U2 -column 1 -row 0

create rp_group rp2 -design mid_design -columns 1 -rows 3
   add_to_rp_group mid_design::rp2 \
      -hierarchy pair_design::rp1 -instance I1 -column 0 -row 0
   add_to_rp_group mid_design::rp2 \
      -hierarchy pair_design::rp1 -instance I2 -column 0 -row 1
   add_to_rp_group mid_design::rp2 \
      -hierarchy pair_design::rp1 -instance I3 -column 0 -row 2
```

Figure 10-18 Instantiating Groups in a Hierarchical Group



Adding Keepouts

To add a keepout to a relative placement group, use the `add_to_rp_group` command. For example, to create a keepout named gap1, enter

```
add_to_rp_group TOP::misc -keepout gap1 \
                -column 0 -row 2 -width 15 -height 1
```

where `TOP::misc` is the group list.

Querying Relative Placement Groups

Table 10-8 lists the commands available for querying particular types of relative placement groups to annotate, edit, and view. For detailed information, see the man pages.

Table 10-8 Commands for Querying Relative Placement Groups

To return collections for this	Use this Command
Relative placement groups that match certain criteria	`get_rp_groups`
All or specified relative placement groups and the included and instantiated groups they contain in their hierarchies	`all_rp_groups`
All or specified relative placement groups that contain included or instantiated groups	`all_rp_hierarchicals`
All or specified relative placement groups that contain included groups	`all_rp_inclusions`
All or specified relative placement groups that contain instantiated groups	`all_rp_instantiations`
All or specified relative placement groups that directly embed leaf cells (added to a group by `add_rp_group -leaf`) or instantiated groups (added to a group by `add_rp_group -hierarchy -instance`) in all or specified relative placement groups in a specified design or the current design	`all_rp_references`

For example, to create a collection of relative placement groups that start with the letter *g* in a design that starts with the letter *r*, enter

```
get_rp_groups r*::g*
{ripple::grp_ripple}
```

To set the utilization to 95 percent for all relative placement groups, enter

```
set_rp_group_options [all_rp_groups] -utilization 0.95
```

Checking Relative Placement Constraints

To check whether the relative placement constraints have been met, run the `check_rp_groups` command. The command reports relative placement failures such as the following:

```
RPGP-028: The height '%f' of the RP group '%s' is more than the height
'%f' of the core area.
RPGP-035 (warning) Relative placement leaf cell data may have been lost.
RPGP-0348: All RP cells in cluster '%s' are not within same voltage
region;
ignoring voltage region placement.
```

The `check_rp_groups` command checks for the following failures:

- The relative placement group cannot be placed as a whole.
- The height or width of the relative placement group is greater than the height or width of the core area.
- The user-specified orientation cannot be met.

If a failure prevents the group from being placed as a single entity, as defined by the relative placement constraints, the failure is considered critical. If the failure does not prevent placement but causes the relative placement constraints to be violated, the failure is considered noncritical. The generated report contains separate sections for critical and noncritical failures.

Note that the `check_rp_groups` command does not check for the failure: the keepouts are not created correctly in the relative placement group.

You can check all relative placement groups by specifying the `-all` option or you can specify which relative placement groups to check. By default, the report is output to the screen. To save the generated report to a file, specify the file name by using the `-output` option.

For example, to check the relative placement constraints for groups compare17::seg7 and compare17::rp_group3 and to save the output in a file called rp_failures.log, run the following command:

```
check_rp_groups "compare17::seg7 compare17::rp_group3" \
             -output rp_failures.log
```

The generated report is similar to this:

```
****************************************
Report : The RP groups, which could not be placed.
Version: 2005.12
Date   : Tue Dec 27 16:32:02 2005
No. of RP groups:1
****************************************
RP GROUP: compare17::seg7
```

```
------------------------------
ERROR: Could not get clean area for placing RP group compare17::seg7.

****************************************
Report : The RP groups, not meeting all constraints but placed.
Version: 2005.12
Date   : Tue Dec 27 16:32:02 2005
No. of RP groups:1
****************************************
RP GROUP: compare17::rp_group_3
------------------------------
WARNING: Could not set user specified orientation for cell u[8].
```

To display a detailed report, use the `-verbose` option, as shown:

```
check_rp_groups -all -verbose
```

The `-verbose` option reports all possible relative placement failures in the design.

Saving Relative Placement Information

You can use the `write_rp_groups` command to write out relative placement constraints.

For example, to save all the relative placement groups to disk, remove the information from the design, and then re-create the information on the design, enter

```
get_rp_groups
{mul::grp_mul ripple::grp_ripple example3::top_group}
write_rp_groups -all -output my_groups.tcl
1
remove_rp_groups -all -quiet
1
get_rp_groups
Error: Can't find objects matching '*'. (UID-109)
source my_groups.tcl
{example3::top_group}
get_rp_groups
{example3::top_group ripple::grp_ripple mul::grp_mul}
```

By default, the `write_rp_groups` command writes out commands for creating the specified relative placement groups and to add leaf cells, hierarchical groups, and keepouts to these groups. The commands for generating subgroups within hierarchical groups are not written. The `write_rp_groups` command writes out updated relative placement constraints and includes names changes from uniquification or ungrouping.

If you specified multiple column positions or multiple row positions for a cell using the `-num_columns` or `-num_rows` options with the `add_to_rp_groups` command, the `write_rp_groups` command writes out the multiple-location cell as well.

You can modify the default behavior of the write_rp_groups command by using the options described in Table 10-9.

Table 10-9 Using the write_rp_groups Command Options

To do this	Use this option with the write_rp_groups Command
Write all the relative placement groups within the hierarchy of the relative placement groups. If you omit this option, subgroups are not written.	-hierarchy
Write only create_rp_group commands to the script.	-create
Write only add_to_rp_group -leaf commands to the script.	-leaf
Write only create_rp_group -keepout commands to the script.	-keepout
Write only create_rp_group -hierarchy -instance commands to the script.	-instance
Write only create_rp_group -hierarchy commands to the script.	-include

Removing Relative Placement Group Attributes

To remove relative placement groups, run the remove_rp_groups command. You can remove all relative placement groups (by specifying the -all option), or you can specify which relative placement groups to remove. If you specify a list of relative placement groups, only the specified groups (and not groups included or instantiated within the specified group) are removed. To remove the included and instantiated groups of the specified groups, you must specify -hierarchy.

For example, to remove the relative placement group named grp_ripple and confirm its removal, enter

```
get_rp_groups
{mul::grp_mul ripple::grp_ripple example3::top_group}
remove_rp_groups ripple::grp_ripple
Removing rp group 'ripple::grp_ripple'
1
get_rp_groups *grp_ripple
Error: Can't find object 'grp_ripple'. (UID-109)
remove_rp_groups -all
Removing rp group 'mul::grp_mul'
```

```
Removing rp group 'example3::top_group'
1
```

To remove relative placement group attributes, run the `remove_rp_group_options` command. You must specify the group name and at least one option; otherwise, this command has no effect.

For example, to remove the `x_offset` attribute for the block1::misc1 group, enter

```
remove_rp_group_options block1::misc1 -x_offset
{block1::misc1}
```

The command returns a collection of relative placement groups for which attributes have been changed. If no attributes change for any object, an empty string is returned.

To remove objects from a relative placement group, use the `remove_from_rp_group` command. You can remove leaf cells (`-leaf`), included groups (`-hierarchy`), instantiated groups (`-hierarchy -instance`), and keepouts (`-keepout`).

For example, to remove leaf cell carry_in_1 from grp_ripple, enter

```
remove_from_rp_group ripple::grp_ripple -leaf carry_in_1
1
```

If you specified multiple column positions or multiple row positions for a cell using the `-num_columns` or `-num_rows` options with the `add_to_rp_groups` command, the `remove_from_rp_group` command removes the cell from all its locations.

Sample Script for a Relative Placement Flow

Example 10-40 is a sample script for running a relative placement flow.

Example 10-40 Sample Script for the Relative Placement Flow

```
# Set library and design paths
source setup.tcl
read_ddc design_name
current_design top
link

# Create relative placement constraints
create_rp_group grp_ripple -design ripple -rows 8
...
add_to_rp_group ripple::grp_ripple -leaf carry_in_1
...

# Apply design constraints
source constraints.tcl

# Read physical constraints
extract_physical_constraints top.def
```

```
# Check relative placement
check_rp_groups

# Perform synthesis
compile_ultra -scan

# Write out the netlist and relative placement
change_names -rules verilog
write -f verilog -h -o ripple.v
write_rp_groups -all -out ripple.rp.tcl
```

Placement Options: Magnet Placement

Magnet placement improves timing and congestion correlation between Design Compiler and IC Compiler. Magnet placement moves standard cells closer to objects specified as magnets. You can use magnet placement with any netlist that is fully mapped. Use the `magnet_placement` command to enable magnet placement if `magnet_placement` was used in IC Compiler.

The tool will only perform magnet placement if the standard cells are placed. If you use the `magnet_placement` command before the standard cells are placed, the tool will not pull any objects to the specified magnet. The tool will issue a warning for this condition.

To perform magnet placement, use the `magnet_placement` command with a specification of the magnets and options for any special functions you need to perform. Table 10-10 lists options to use for various tasks. Specify magnet placement as follows:

magnet_placement [*options*] *magnet_objects*

For complete command syntax and description details, see the man page.

Table 10-10 Using the magnet_placement Command Options

To do this	Use this option
Enable the movement of fixed cells and display a warning that lists the fixed cells to be moved	`-move_fixed`
Fix cells in their locations after magnet placement	`-mark_fixed`
Specify the number of logic levels from the magnet that should be checked for magnet placement	`-logical_level` *number*
Prevent movement of buffers and inverters	`-exclude_buffers`

Table 10-10 Using the magnet_placement Command Options (Continued)

To do this	Use this option
Prevent cells from being placed over soft blockages	-avoid_soft_blockages
Prevent placement beyond sequential cells	-stop_by_sequential_cells
Pull only those objects on the timing path between the magnet object and a specified end object. Note: This option is mutually exclusive with -logical_level.	-stop_points *object_list*

Use the magnet_placement_fanout_limit variable to specify the fanout limit. If the fanout of a net exceeds the specified limit, the magnet_placement command does not pull cells of the net toward the magnet objects. The default setting is 1000.

By default, the magnet placement operation is terminated before the sequential cell when the -stop_by_sequential_cells option is used with the magnet_placement command. If you want to terminate the magnet placement operation after the sequential cell, set the magnet_placement_stop_after_seq_cell variable to true. The default value is false.

Magnet placement allows cells to be overlapped by default. To prevent overlapping of cells, you can set the magnet_placement_disable_overlap variable to true.

To return a collection of cells that can be moved with magnet placement, use the get_magnet_cells command with the options you need. Table 10-11 lists options to use for various tasks. For command syntax and details, see the man page.

get_magnet_cells [*options*] *magnet_list*

Resetting Physical Constraints

To reset all physical constraints, use the reset_physical_constraints command. Use this command to clear all existing physical constraints before reading in a new or modified floorplan.

Saving Physical Constraints Using the write_floorplan Command

You use the write_floorplan command to save the floorplan information. The write_floorplan command writes out individual floorplan commands relative to the top of the design, regardless of the current instance. The output is a command script file that contains floorplan information such as bounds, placement blockages, route guides, plan groups, and voltage areas.

If you reuse this output to re-create the floorplan, you must read it in from the top level of the design with the read_floorplan command.

You can also use the source command to import the floorplan information. However, the source command reports errors and warnings that are not applicable to DC Ultra in topographical mode. The read_floorplan command removes these unnecessary errors and warnings. In addition, you need to enable fuzzy name matching manually when you use the source command. The read_floorplan command automatically enables fuzzy name matching.

Note:
The output of the DC Ultra write_floorplan command does not contain all the physical information; it only contains the physical data that is applicable to DC Ultra topographical mode, which is a subset of the entire physical data set. Also, physical data, such as vias, is transformed to pre-routes when read into DC Ultra.

For command syntax and other details, see the man page.

Reporting Physical Constraints

To report the physical constraints, use the report_physical_constraints command. To report the create_net_shape commands (preroutes), use the -pre_route option the report_physical_constraints command. If you do not want the report to show site row information, use the -no_site_row option.

Performing Automatic High-Fanout Synthesis

By default, Design Compiler topographical mode does automatic high-fanout synthesis. Design Compiler does not perform automatic high-fanout synthesis on any nets that are part of dont_touch_network and ideal_network. You can use the set_ahfs_options command to specify the constraints to be used when running automatic high-fanout synthesis (AHFS). After automatic high-fanout synthesis, Design Compiler sets a global design-based attribute that prevents IC Compiler from doing automatic high-fanout synthesis. If you need to enable automatic high-fanout synthesis in IC Compiler, for example, if you are changing any ideal or dont_touch_network settings, apply the set_ahfs_options command in the back-end script.

Test Synthesis in Topographical Mode

Topographical mode supports test synthesis. The flow is similar to the one in Design Compiler wire load mode except that the insert_dft command is used for stitch-only. Topographical mode supports basic scan and adaptive scan.

To perform scan insertion within topographical mode, use a script similar to the following:

```
dc_shell -topo
read_ddc top_elaborated.ddc
source top_constraint.sdc
source physical_constraints.tcl
compile_ultra -gate_clock -scan

## Provide DFT specifications
set_dft_signal ...

create_test_protocol
dft_drc
preview_dft
insert_dft
dft_drc

compile_ultra -incremental -scan
write_scan_def -output dft.scandef
```

For more information on scan insertion, see the DFT Compiler documentation.

Using Power Compiler in Topographical Mode

In topographical mode, Design Compiler can perform power correlation. The `set_power_prediction` command enables Design Compiler to correlate post-synthesis power numbers with those after place and route. Optionally, you can use the `-ct_references` option to specify clock tree references to improve correlation.

In addition, power prediction is on by default if any gate-level power optimizations are enabled by using the following commands:

- `set_max_dynamic_power`
- `set_max_leakage_power`
- `compile_ultra -gate_clock`

To perform power correlation within topographical mode, run a script similar to the following one:

```
read_ddc top_elaborated.ddc

## Define clock gating style. The default style suits most designs.
Change
only if necessary.

#set_clock_gating_style -pos {integrated} \
     -control_point before

source top_constraint.sdc
source physical_constraints.tcl

## Set power optimization constraints

set_max_leakage_power 0 mW
compile_ultra -gate_clock -scan
write -format ddc -output synthesized.ddc
```

For more information on power correlation, see the Power Compiler documentation.

Multivoltage Designs

Design Compiler topographical mode supports the use of multivoltage designs, multivoltage design features, and related commands.

For multivoltage designs, the subdesign instances (blocks) operate at different voltages. To reduce power consumption, multivoltage designs typically make use of power domains. The blocks of a power domain can be powered up and down, independent of the power state of other power domains (except where a relative always-on relationship exists between two power domains). In particular, power domains can be defined and level shifter and isolation cells can be used as needed to adjust voltage differences between power domains and to isolate shut-down power domains.

A power domain is defined as a logic grouping of one or more hierarchical blocks in a design that share the following:

- Primary voltage states or voltage range (that is, the same operating voltage)
- Power net hookup requirements
- Power-down control and acknowledge signals (if any)
- Power switching style
- Same process, voltage, and temperature (PVT) operating condition values (all cells of the power domain except level shifters)
- Same set or subset of nonlinear delay model (NLDM) target libraries

Principal power domain commands are described in “Specifying Power Intent” on page 6-17.

Note:
Power domains are not voltage areas. A power domain is a grouping of logic hierarchies, whereas the corresponding voltage area is a physical placement area into which the cells of the power domain’s hierarchies are placed. This correspondence is not automatic. You are responsible for correctly aligning the hierarchies to the voltage areas. You use the `create_voltage_area` command to set voltage areas.

Because there are nets that cross power domains (connecting cells operating at different voltages) and because some power domains can be always-on (that is, never powered down) while others might be always-on relative to some specific power domain (requiring isolation) and still others shut down and power up independently (also requiring isolation), special cells are needed.

In general, voltage differences are handled by level shifters, which connect drive and load pins operating at different voltages across the power domains. They are used to step up or step down the voltage from their input side to their output side. These cells are modeled

either as simple buffers or as buffer cells with an enable pin. The first type of cell is referred to as a buffer-type level shifter and the second type as an enable-type level shifter. Enable-type level shifters are used when a power domain must be selectively shut down for some duration of the design's operation.

Isolation cells are used when a power domain must be selectively isolated from other power domains at certain times during the design's operation but stepping the voltage from the input side to the output side of the cell is not necessary. Their function is equivalent to that of an enable-type level shifter that connects two power domains operating at the same voltage. (Because an enable-type level shifter is basically a buffer, it can connect drive and load pins operating at the same voltage as well as pins not operating at the same voltage.)

Level shifters and isolation cells are not usually part of the original design description and must be inserted during the logic synthesis flow. Different methods are used to add these special cells to a design. Buffer-type level shifters, enable-type level shifters, and isolation cells can be inserted in the following ways:

- Automatically, as part of running the `compile_ultra` command.

  This method is described in "Specifying Power Intent" on page 6-17 and is the recommended method.

- Manually

  This method involves instantiating the cells in the RTL.

The compile flow for multivoltage designs can involve using a number of commands not discussed here. For details, see the *Power Compiler User Guide*.

Compile Flows in Topographical Mode

In addition to supporting a top-down `compile_ultra` flow, as described in the "Top-Down Compile" section in Chapter 8, "Optimizing the Design", DC Ultra topographical mode supports incremental and hierarchical flows, as described in the following sections:

- Performing an Incremental Compile

  The `-incremental` option of the `compile_ultra` command allows you to employ a second-pass, incremental compile strategy.

- Performing a Bottom-up or Hierarchical Compile

  Topographical mode supports a hierarchical flow or bottom-up flow if you need to address design and runtime challenges or use a divide and conquer synthesis approach. In topographical mode, the `-top` option of the `compile_ultra` command enables you to stitch compiled physical blocks into the top-level design.

Performing an Incremental Compile

The `-incremental` option of the `compile_ultra` command allows you to employ a second-pass, incremental compile strategy. The main goal for `compile_ultra -incremental` is to enable topographical-based optimization for post-topographical-based synthesis flows such as retiming, design-for-test (DFT), DFT MAX, and minor netlist edits. The primary focus in Design Compiler topographical mode is to maintain QoR correlation; therefore, only limited changes to the netlist can be made.

Use the incremental compile strategy to meet the following goals:

- Improve design QoR
- Fix the netlist after manual netlist edits or constraint changes
- Fix the netlist after various synthesis steps have been performed on the compiled design, for example, after `insert_dft` or register retiming
- Control design rule fixing by using the `-no_design_rule` or `-only_design_rule` option in combination with the `-incremental` option

Incremental compile supports adaptive retiming, that is, `compile_ultra -incremental -retime`.

Note that applying `compile_ultra -incremental` to a topographical netlist results in placement-based optimization only. This compile should not be thought of as an incremental mapping.

Note:
When you use the `insert_buffer` command and `remove_buffer` command described in "Editing Designs" on page 5-31, the `report_timing` command does not report placement-based timing for the edited cells. To update timing, run the `compile_ultra -incremental` command.

When using the `-incremental` option, keep the following in mind:

- Marking library cells with the `dont_use` attribute does not work for an incremental flow when it is applied to a topographical netlist. Make sure to apply any `set_dont_use` attributes before the first pass of a topographical-based synthesis.
- If you intend to use boundary optimization and scan insertion, apply them to the first pass of a topographical-based synthesis.
- Avoid significant constraint changes in the incremental pass.

  Note:
  Physical constraint changes are not supported.

Performing a Bottom-up or Hierarchical Compile

In a bottom-up or hierarchical compile, you compile the subdesigns separately and then incorporate them in the top-level design.

The recommended strategy is a top-down compile flow. However, topographical mode supports a hierarchical flow or bottom-up flow if you need to address design and runtime challenges or use a divide and conquer synthesis approach. In the bottom-up strategy, individual subblocks are constrained and compiled separately. The compiled subblocks are then included in subsequent top-level synthesis. In topographical mode, the tool can read two types of hierarchical blocks:

- Interface logic model generated in topographical mode or IC Compiler
- Netlist generated in topographical mode

That is, you can compile the subblock in topographical mode and provide it to the top-level design as a .ddc netlist or as an interface logic model (ILM). Alternatively, you can continue working on the subblock in IC Compiler to create an interface logic model, which you can then provide to the top-level design in topographical mode. Timing and physical information of the subblock are propagated to the top-level for physical synthesis in topographical mode. In addition, you can provide placement constraints for a subblock during top-level synthesis to maintain correlation to IC Compiler.

The `-top` option of the `compile_ultra` command enables you to perform top-level design integration—that is, you can stitch compiled physical blocks into the top-level design without losing placement-aware net delay estimation for register-to-register paths within lower-level blocks. The tool automatically back-annotates the top-level design with virtual placement-based delays of subblocks while preserving the logic structure of the subblocks. However, top-level design integration does not use core placement technology to drive mapping and optimization at the top level. In addition, every subblock must be fully mapped.

In the hierarchical or bottom-up flow, the tool propagates block-level timing and placement to the top level and uses them to drive optimizations. In addition, you can specify location constraints for the subblock. Top-level optimizations are placement-aware and can be driven with the same physical constraints as your back-end tool.

The following sections describe the DC Ultra topographical mode hierarchical flow:

- Overview of Bottom-Up Compile
- Compiling the Subblock
- Compiling the Design at the Top Level
- Performing Top-Level Design Stitching
- Steps in the Top-Level Design Stitching Flow

Overview of Bottom-Up Compile

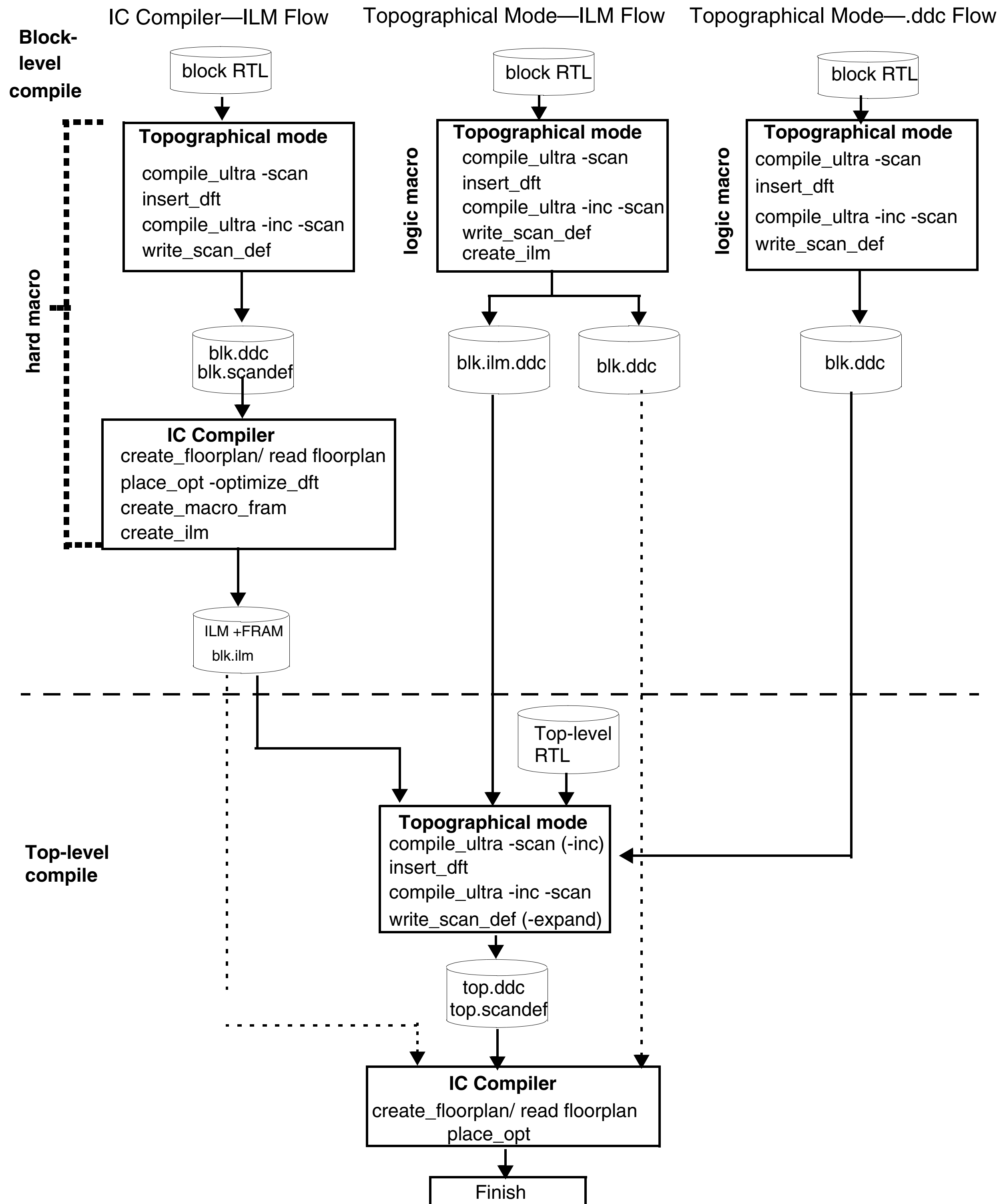
In the bottom-up strategy, you first compile the subblock in topographical mode and save the mapped subblock as a netlist (.ddc format) or an interface logic model (.ddc format). You can then continue working on the subblock (.ddc netlist) in IC Compiler to generate an interface logic model.

Top-level synthesis can accept any of these three types of mapped subblocks: .ddc netlist synthesized in topographical mode, interface logic model created in topographical mode, or interface logic model created in IC Compiler. Therefore, three types of bottom-up flows are supported as shown in Figure 10-19 on page 10-76:

- IC Compiler—interface logic model flow
- Topographical mode—interface logic model flow
- Topographical mode—.ddc flow

Note:
IC Compiler cannot accept an interface logic model created in topographical mode.

Figure 10-19 Overview of the Hierarchical Flow



As shown in Figure 10-19 on page 10-76, the bottom-up flow requires these main steps:

1. Compile the subdesigns independently. See "Compiling the Subblock" on page 10-77.
2. Read in the top-level design and any compiled subdesigns not already in memory; compile the top-level design. See "Compiling the Design at the Top Level" on page 10-79.

Compiling the Subblock

As shown in Figure 10-19 on page 10-76, compiling the subblock requires the following steps:

1. Specify the logical and physical libraries as described in "Specifying Libraries" on page 10-9.
2. Read in the subblock (Verilog, VHDL, netlist, or .ddc) and set the current design to the subblock.
3. Apply block-level timing constraints and power constraints.
4. (Optional) Provide physical constraints as described in "Using Floorplan Physical Constraints" on page 10-15.
5. (Optional) Visually verify the floorplan

   Use the Design Vision layout window to visually verify that your pre-synthesis floorplan is laid out according to your expectations. The layout view automatically displays floorplan constraints read in with `extract_physical_constraints` or read in with Tcl commands. You need to link all applicable designs and libraries to obtain an accurate floorplan. For details, see the *Design Vision User Guide* or Design Vision Online Help.
6. Compile the subblock by using the `compile_ultra -scan` command.
7. Specify the design-for-test configuration and run the `insert_dft` command as described in "Test Synthesis in Topographical Mode" on page 10-68.

   You use the `insert_dft` command to insert scan chains, if the subblock is to be included in the top-level scan chain.
8. Run the `compile_ultra -scan -incremental` command.

   The `-scan` option enables the tool to map sequential cells to their scan-equivalent cells.
9. Write out the following information:
   - Save the SCANDEF information to the block database by using the `write_scan_def` command. The SCANDEF file is a data file that contains scan chain information so that the place and route tools can perform reordering and repartitioning of scan chains.

- Save the mapped subblock in the .ddc format by using the `write -format ddc` command.

10. The steps you perform from this point forward depend on whether you are creating a full .ddc netlist for the subblock in topographical mode, generating an ILM model for the subblock in topographical mode, or continuing to synthesize the subblock in IC Compiler.

    When creating a full .ddc netlist for the subblock in topographical mode, no additional steps are necessary; during top-level synthesis, read in the mapped subblock or add it to the `link_library` variable. See "Compiling the Design at the Top Level" on page 10-79.

    When generating an ILM model for the subblock in topographical mode, follow these steps:

    a. To verify consistency between the ILM model and the original netlist, use the `write_interface_timing` command to create a timing report for the original netlist.

    b. Run the `create_ilm` command to generate the ILM.

       By default, the `create_ilm` command writes out the ILM in the .ddc format; therefore, you do not need to specify the `-format` option.

    c. Use the `write_interface_timing` command to create a timing report for the ILM model. This timing report will be compared to the timing report for the netlist to help verify consistency between the ILM model and the netlist.

    d. Compare the original netlist and ILM interface timing reports by using the `compare_interface_timing` command. This command checks for consistency between the ILM model and the original netlist. For example, the command

       ```
       compare_interface_timing original.wit ilm.wit -output check.cit
       ```

       compares the report for the original netlist in the original.wit file with the report for the ILM model in the ilm.wit file and outputs the comparision report to the check.cit file.

    e. Use the `write -format ddc` command to save the ILM.

    f. During top-level synthesis, read in the interface logic model (.ddc netlist) of the mapped subblock or add it to the `link_library` variable. See "Compiling the Design at the Top Level" on page 10-79.

    When continuing to synthesize the subblock in IC Compiler (the IC Compiler ILM flow), use the following steps to create a physical block. For more information on IC Compiler commands, see the IC Compiler documentation.

    a. In IC Compiler, set up the design and Milkyway libraries.

    b. Read in the .ddc format of the mapped subblock and block SCANDEF information generated in topographical mode.

    c. Create the floorplan by using IC Compiler commands or read in floorplan information from a DEF file by using the `read_def` command.

d. Run the `place_opt -optimize_dft` command to perform placement-aware scan reordering.

e. Use the `save_mw_cel` command to create the CEL view.

f. Use the `create_macro_fram` command to create the FRAM view.

g. Use the `write_interface_timing` command to create a timing report for the original netlist. This timing report will be compared to the timing report for the ILM model to help verify consistency between the ILM model and the netlist.

h. Use the `create_ilm` command to generate the interface logic model for the subblock.

i. Use the `write_interface_timing` command to create a timing report for the ILM model. This timing report will be compared to the timing report for the netlist to help verify consistency between the ILM model and the netlist.

j. Compare the original netlist and ILM interface timing reports by using the `compare_interface_timing` command. This command checks for consistency between the ILM model and the original netlist. For example, the command

```
compare_interface_timing original.wit ilm.wit -output check.cit
```

compares the report for the original netlist in the original.wit file with the report for the ILM model in the ilm.wit file and outputs the comparision report to the check.cit file.

k. During top-level synthesis, read in the interface logic model (.ddc netlist) of the mapped subblock or add it to the `link_library` variable. See "Compiling the Design at the Top Level" on page 10-79.

Compiling the Design at the Top Level

As shown in Figure 10-19, compiling the design at the top level requires the following steps:

1. In topographical mode, set the current design to the top-level design, link the design, and apply the top-level timing constraints.
2. Read in the subblock that you mapped in "Compiling the Subblock" on page 10-77, that is, any of the following:
   - Interface logic model created in IC Compiler. That is, add the Milkyway design library that contains the ILM view to the Milkyway reference library list at the top level. In addition, add the subblock ILM to the `link_library` variable.
   - Interface logic model (.ddc format) created in topographical mode (or add it to the `link_library` variable)
   - Netlist (.ddc) created in topographical mode (or add it to the `link_library` variable).

In this case, you must also specify how the subblock should be treated during top-level synthesis. To specify that top-level synthesis cannot change the logical structure but can change the placement, use the `set_dont_touch` command to set a logical `dont_touch` attribute on the subblock. To specify that the subblock should be treated as a physical subblock—that is, top level synthesis does not modify the physical or logical structure of the subblock—use the `set_physical_hierarchy` command. If you do not specify how the subblock should be handled, top-level synthesis treats it as flat; that is, it can modify both the logical and physical structure.

3. Read in the top-level design (Verilog, VHDL, netlist, or .ddc) file.
4. Apply top-level timing and power constraints.
5. (Optional) Provide physical constraints as described in "Using Floorplan Physical Constraints" on page 10-15.

   You can specify locations for the subblock by using the `set_cell_location` command or by using the `extract_physical_constraints` command to extract physical information from the Design Exchange Format (DEF) file.
6. (Optional) Visually verify the floorplan

   Use the Design Vision layout window to visually verify that your pre-synthesis floorplan is laid out according to your expectations. The layout view automatically displays floorplan constraints read in with `extract_physical_constraints` or read in with Tcl commands. You need to link all applicable designs and libraries to obtain an accurate floorplan. For details, see the *Design Vision User Guide* or Design Vision Online Help.
7. Run the `compile_ultra -scan` command.

   The `-scan` option enables the tool to map sequential cells to appropriate scan flip-flops.
8. Run the `insert_dft` command to insert scan chains at the top level, followed by `compile_ultra -scan -incremental` command.
9. Write out the top-level netlist by using the `write -format ddc` command.

   Note:
   In ILM-based flows, you must remove all subblock ILMs before generating the final top-level netlist.
10. Write out the SCANDEF information by using the `write_scan_def` command.

    Top-level scan chain stitching connects scan chains up to the interfaces of test pins of the subblock. In the top-level SCANDEF, you can control whether physical blocks are described using the "BITS" construct or scan chain object.

In the topographical mode—ILM and .ddc flows, you use the `write_scan_def` command with the `-expand_elements` option to write out the SCANDEF information. You do not maintain the test hierarchy partition during top-level physical implementation in IC Compiler. The SCANDEF must directly reference the scan leaf objects of the physical block instead of the BITS construct to run the IC Compiler top-level flat flow. For example,

```
write_scan_def -expand_elements block_instance
```

In the IC Compiler—ILM flow, you maintain the test hierarchy partition during top-level physical implementation. Therefore, the top-level SCANDEF is written out with the block containing the BITS construct; you use the `write_scan_def` command without the `-expand_elements` option to write out the SCANDEF information.

Performing Top-Level Design Stitching

In topographical mode, the `-top` option of the `compile_ultra` command enables you to stitch compiled physical blocks into the top-level design.

The recommended strategy is a top-level compile flow. However, you might need to use the top-level design stitching capability in certain cases. For example,

- The chip top level might contain unmapped glue logic that needs to be mapped, timed, and optimized.
- Paths that traverse multiple physical blocks might require the complete timing context at the chip level.

The `-top` option of the `compile_ultra` option enables you to perform chip-level mapping, optimization, and stitching. Top-level design stitching in topographical mode is similar to the functionality provided by the `compile -top` command except that in topographical mode, the tool can map any unmapped top-level glue logic. The tool automatically back-annotates the top-level design with virtual placement-based delays of subblocks while preserving the logic structure of the subblocks.

When you perform top-level design stitching, keep the following points in mind:

- Each subblock must be a fully mapped netlist. Otherwise, the following error message is issued:

  ```
  Error: Using compile -top with unmapped logic (OPT-1304)
  ```

  The subblock must have been compiled in topographical mode. Otherwise, the following warning message is issued:

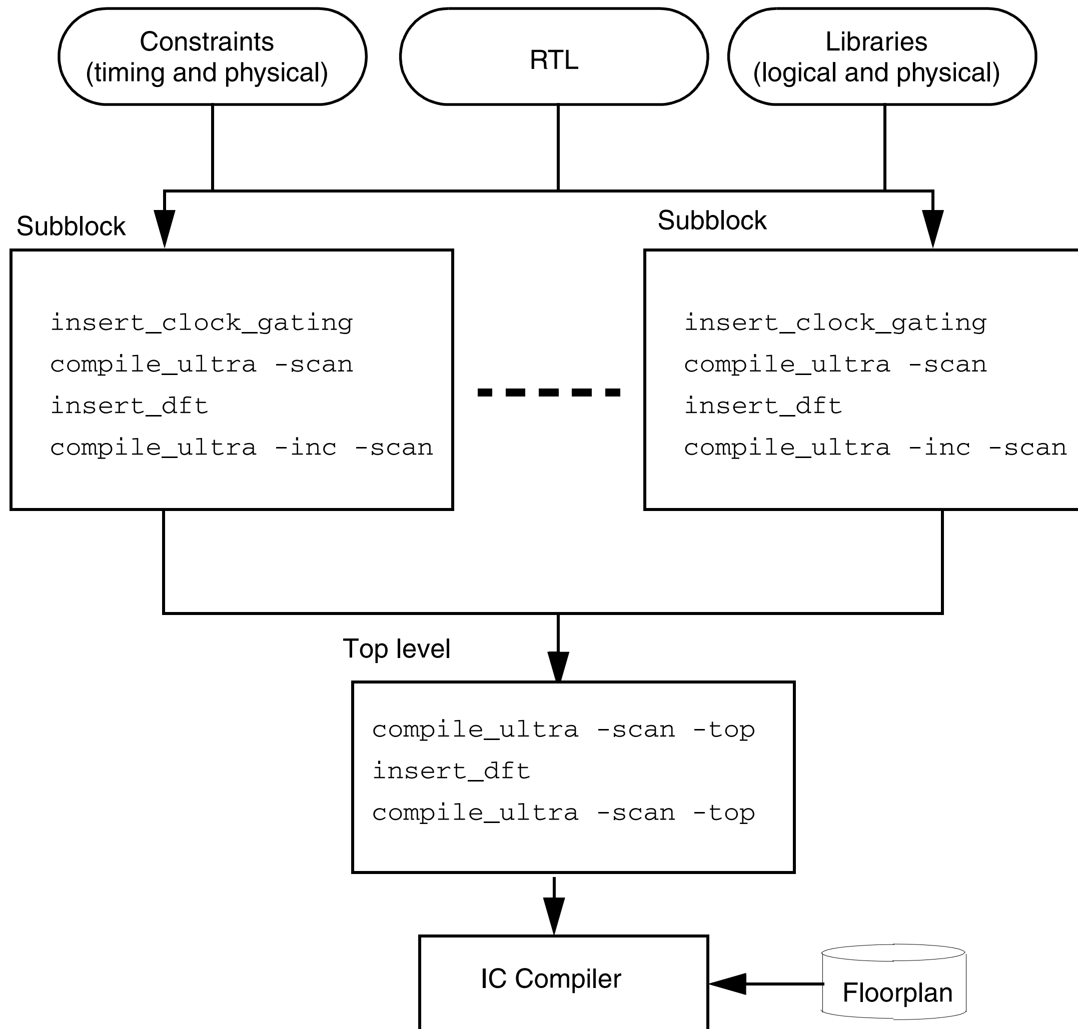
  ```
  Warning: Subblock '%s' was not created using DC
  topographical technology. (OPT-1422)
  ```

- If the subblock is to be included in the top-level scan chain, use the `insert_dft` command to insert scan chains in the subblock.

Steps in the Top-Level Design Stitching Flow

Figure 10-20 shows the steps in the top-level design stitching flow.

Figure 10-20 Top-Level Design Stitching Flow



As shown in Figure 10-20, the top-level design stitching flow requires these steps:

1. Set up the design and libraries.
2. Read in the top-level design.
3. Compile each subblock by performing the following steps:
   a. Set the current design to the sub-block.
   b. Apply timing constraints and power constraints.

c. (Optional) Provide physical constraints as described in "Using Floorplan Physical Constraints" on page 10-15.

d. Perform clock gating by using the `insert_clock_gating` command.

e. Perform test-ready compile by using the `compile_ultra -scan` command.

f. If the subblock is to be included in the top-level scan chain, use the `insert_dft` command to insert scan chains.

g. Run the `compile_ultra -scan -incremental` command.

h. Use the `report_timing` command to check timing.

4. Set the current design to the top-level design, link the design, and apply the top-level timing constraints.

5. Perform clock gating at the top level by using the `insert_clock_gating` command.

6. Run the `compile_ultra -scan -top` command.

   You use these options because top-level glue logic can contain both unmapped sequential cells and combinational cells. The `-top` option maps the top-level logic and back-annotates the top-level design with timing information from the lower-level blocks. The `-scan` option enables the tool to map sequential cells to appropriate scan flip-flops.

7. Run the `insert_dft` command to insert scan chains at the top level, followed by the `compile_ultra -scan -top` command to map any additional unmapped logic that might have been introduced.

Note:

The `compile_ultra -top` command performs boundary optimizations that could change interface logic between blocks; therefore, the back-annotated timing of subblocks might be updated even though the logic configuration has not changed.

The top-level design stitching feature does not support clock tree estimation; therefore, the `report_power` command cannot report any correlated power at the top level.

Supported Commands, Command Options, and Variables

In topographical mode, if an unsupported command or variable is encountered in a script, an error message is issued; however, the script *continues*.

For example,

```
ERROR: Command set_wire_load_mode is not supported in DC
Topographical mode. (OPT-1406)
```

For unsupported command options (except wire load model options), an error message is issued, and the script stops executing. Wire load model options are ignored, and the script continues.

Also, variables that are read-only in topographical mode are indicated read-only variables in error messages.

Look for additional supported commands, command options, and variables in subsequent releases. You should check your scripts and update them as needed. See the appropriate man pages to determine the current status in topographical mode.

Using the Design Compiler Graphical Tool

The Design Compiler Graphical tool extends topographical technology and enables you to optimize multicorner-multimode (MCMM) designs, reduce routing congestion, and improve both area correlation with IC Compiler and runtime in IC Compiler by using Synopsys physical guidance. In addition, Design Compiler Graphical enables you to create and modify floorplans using floorplan exploration.

This section describes Design Compiler Graphical features in following subsections:

- Reducing Routing Congestion
- Improving Area Correlation and Runtime with Synopsys Physical Guidance (SPG)
- Creating and Modifying Floorplans Using Floorplan Exploration

For multicorner-multimode support details, see “Optimizing Multicorner-Multimode Designs in Design Compiler Graphical” on page 10-102.

Reducing Routing Congestion

This section describes how to reduce routing congestion in the following subsections:

- Routing Congestion Overview
- Viewing Congestion With the Design Vision Layout Window
- Routing Congestion Reduction Flow
- Predicting, Analyzing, and Minimizing Routing Congestion
- Specifying Congestion Optimization Options
- Reporting Congestion
- Summary of Routing Congestion Commands

Routing Congestion Overview

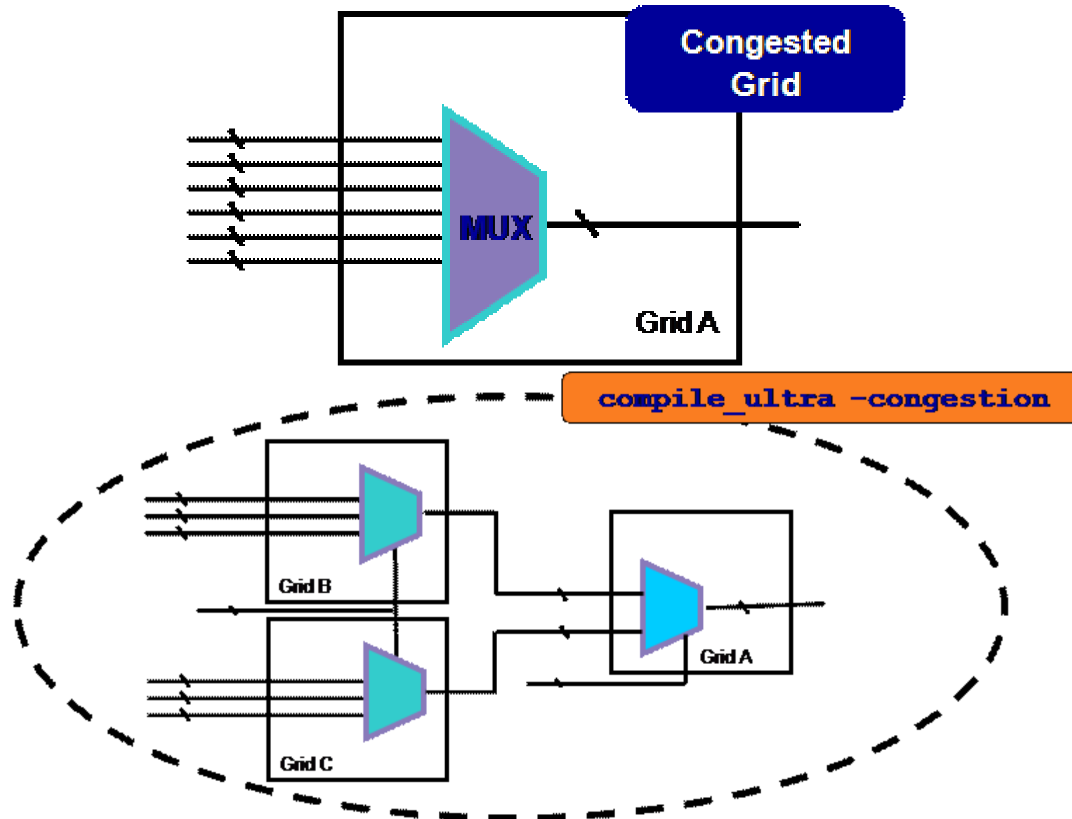
Designs are considered congested when the wires needed to connect design components become greater than the space available to put the wires. If Design Compiler passes a congested design to IC Compiler, it might be difficult and time consuming for IC Compiler to reduce the congestion. Design Compiler can reduce congestion easier than IC Compiler because Design Compiler can optimize RTL structures. Traditionally, to minimize congestion related to the netlist topology, you would use scripting solutions to restrict technology mapping tools, such as IC Compiler, from choosing library cells that you perceive as poor for congestion. But these solutions do not significantly improve congestion; instead, they often create poor timing or area results.

To reduce congestion in Design Compiler, use the `-congestion` option of the `compile_ultra` command. In addition, you can use the `set_congestion_options` command to specify options such as the amount of resources available for a given layer. To report details about congestion, use the `report_congestion` and the `report_congestion_options` commands. For graphical representation of the congested areas, use the Design Vision layout window. From this window you can view and analyze congestion.

You use the congestion optimization feature primarily for designs in which RTL logic structures result in congestion. This standard cell congestion is often caused by the topology of the netlist. Minimizing the congestion for such designs often requires significant changes in the netlist topology, which cannot be done during place and route. Such large changes are best done early in the flow during synthesis. For best results, start your congestion reduction at the RTL level not at the gate-level netlist. If needed, the `compile_ultra -congestion` command does support gate-level congestion-driven synthesis optimizations.

When you run the `compile_ultra -congestion` option, the tool attempts to produce logic structures that are better for placement and congestion. Figure 10-21 shows an example of how the tool optimizes an RTL structure to minimize congestion. In this example, the tool optimizes the single large multiplexor into three smaller multiplexors which reduces routing congestion by restructuring the RTL.

Figure 10-21 Optimizing RTL Structures to Minimize Congestion



Viewing Congestion With the Design Vision Layout Window

The Design Vision layout window enables you to view and analyze the physical aspects of a design. The layout view displays physical information such as placement area, placement blockages, ports, macros, selected standard cells and congestion maps. You can use the layout window to do the following:

1. Validate physical constraints that you have applied in topographical mode.
2. Debug QoR issues related to floorplan and topographical mode layout.
3. Perform GUI-based congestion analysis.

The layout window includes visual modes that provide a high-level view of the placement quality of logic blocks and hierarchical cells in your physical design. Visual modes highlight design information in colored overlays on the active layout view.

- In the hierarchy visual mode, you can color all the cells on a particular hierarchy level or just the hierarchical cells that you select.
- In the snapshot visual mode, you can select and color individual design objects or groups of objects.

For more information, see the Design Vision Help System.

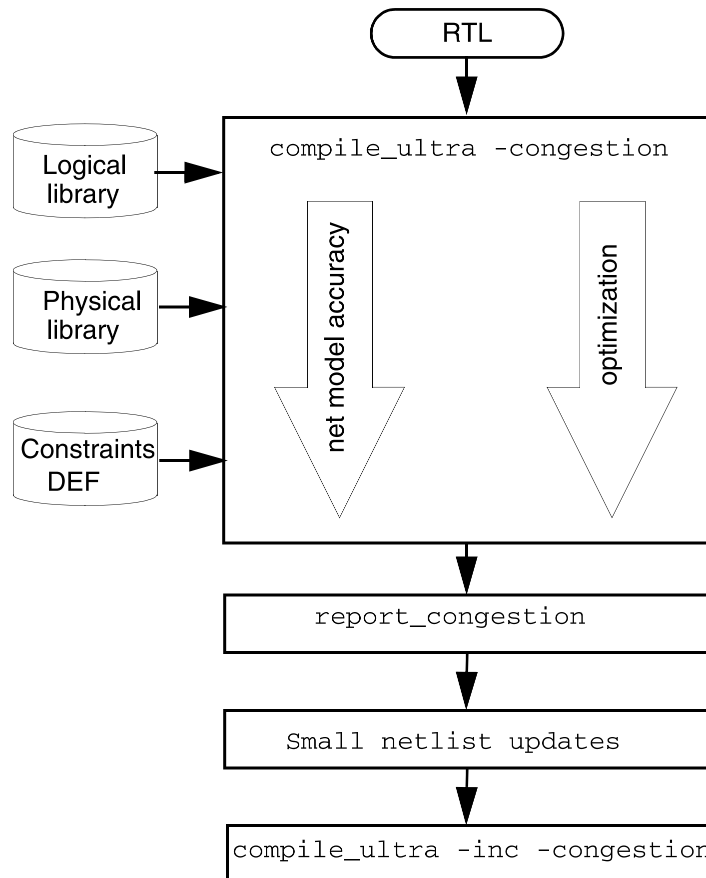
In some cases, it may be useful to open the layout window and display the congestion map or a visual mode images with your script. You can also save an image of the layout in file. For more information, see the *Design Vision User Guide*.

For more information on the Design Vision layout view, see the Design Vision Online Help.

Routing Congestion Reduction Flow

You use the `-congestion` option of the `compile_ultra` command to minimize congestion. Figure 10-22 shows how the congestion optimization feature fits into the synthesis flow. It is recommended that you run this command on the RTL.

Figure 10-22 Congestion Optimization Flow in Design Compiler Graphical



Predicting, Analyzing, and Minimizing Routing Congestion

You use Design Compiler Graphical to predict wire-routing congestion during synthesis, similar to how you predict timing, area, power, and testability in DC Ultra topographical mode. You can predict wire-routing congestion "hot spots" during RTL synthesis and use its interactive capability to visualize the design's congestion. After you have determined that the design has congestion problems, you can minimize congestion by enabling the congestion optimization feature, resulting in a better starting point for layout.

Wire-routing congestion in a design occurs when the number of wires traversing a particular region is greater than the capacity of the region. If the ratio of usage-to-capacity is greater than 1, the region is considered to have congestion problems. Congestion can be associated with standard cells or the floorplan. Typically, standard cell congestion is caused

by the netlist topology, for example, large multiplexer trees, large sums of products (ROMs), test decompression logic, or test compression logic. Floorplan congestion can be caused by macro placement or port location.

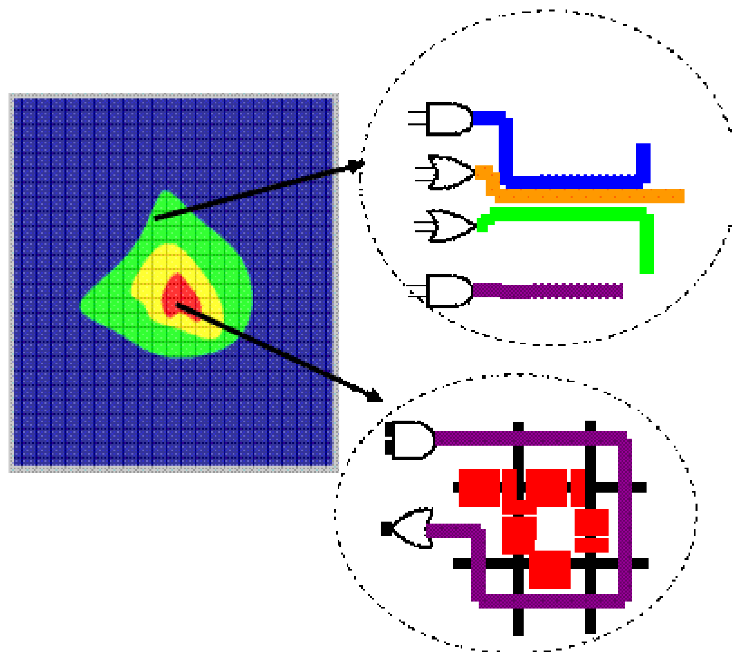
Design Compiler Graphical uses the Synopsys virtual global routing technology to predict wire-routing congestion. In this technology, congestion is estimated by dividing the design into a virtual grid of global routing cells followed by a global route to count the number of wires crossing each grid edge. The capacity and size of each global routing cell is calculated by using the technology physical library information. A cell is considered congested if the number of wires passing through it is greater than the number of available tracks.

Design Compiler Graphical predicts the congestion number by subtracting the maximum wires allowed for a particular edge direction from the total number of wires crossing per edge. Any number over zero is a congestion violation. This calculation is represented by the following equation:

```
Congestion = (number of wires) - (maximum wires allowed)
```

Figure 10-23 shows how Design Compiler Graphical predicts congestion. The red areas indicate congestion.

Figure 10-23 Predicting Congestion Using Virtual Grid Route



The tool considers floorplan-related physical constraints when it estimates routing congestion. In particular, the wiring keepout physical constraint ensures that the tool is aware of areas that must be avoided during routing. This physical constraint helps to achieve consistent congestion correlation with layout.

Specifying Congestion Optimization Options

To specify options for congestion optimization, use the `set_congestion_options` command. This command helps achieve consistent congestion correlation between DC Ultra topographical mode and IC Compiler by ensuring that both tools see the same congestion setup.

To remove congestion options, use the `remove_congestion_options` command.

To report congestion options, use the `report_congestion_options` command.

Use the `set_congestion_options` command options as described below:

Table 10-11 Using the set_congestion_options Command

To do this	Use this option
Minimize congestion by moving cells from congested regions to uncongested regions. Note: Use the `-max_util` option to specify the maximum utilization the tool allows as cells migrate into uncongested areas. For example, setting a value of 0.9 allows the tool to place cells in an area with up to 90 percent utilization.	`-max_util`
Specify the layer whose availability you want to reduce.	`-layer`
Specify how much of the routing resource for the given layer is available. A value of 0.30 for availability means the tool considers 70% as being used.	`-availability`
Specify the lower left and upper right coordinates for which the congestion options apply.	`-coordinate`

Reporting Congestion

The text based `report_congestion` command provides a quick estimate of the congestion status of the design. In cases where the design is reported as significantly congested, you can further analyze congestion by generating a congestion map. By viewing the congestion map, you can identify areas of high congestion and determine whether the design can be routed.

For this detailed analysis, you use the Design Vision layout view to analyze the congestion map. You load the .ddc netlist, synthesized in topographical mode to analyze the congestion map and identify the likely cause of congestion in the design. For more information, see the Design Vision Online Help.

The `report_congestion` command generates a report that shows the design's congestion status. You can use this report to assess the severity of congestion in the design. Example 10-41 shows a sample output generated by the `report_congestion` command.

Example 10-41 The report_congestion Command Output

```
****************************************
Report : congestion
Design : Lut
Version: B-2008.09
Date   : Wed July 20 17:38:50 2007
****************************************
Both Dirs: Overflow = 4651 Max = 5 (10 GRCs) GRCs = 3401 (0.59%)
H routing: Overflow = 2386 Max = 3 (50 GRCs) GRCs = 1788  (0.31%)
V routing: Overflow = 2265 Max = 3 (18 GRCs) GRCs = 1961  (0.34%)
```

You interpret this report as follows:

- *Overflow* is the total number of wires in the design global routing cells that do not have a corresponding track available. The following equation shows how overflow is measured:

$$\text{Overflow} = \sum_{i=1}^{\text{Max}} \text{violation(i)} \times \text{GRCs with violation(i)}$$

- *Max* corresponds to the highest number of over-utilized wires in a single global routing cell. It is the worst-case violation and the number of global routing cells that have the violation. The report shows that the worst-case violation is five and it occurs for ten global routing cells in the design.
- *GRC* is the total number of over-congested global routing cells in the design. The report indicates that there are 3401 violating GRCs in the design, which account for 0.59% of the design.

Summary of Routing Congestion Commands

Commands related to routing congestion reduction are listed below:

- `compile_ultra -congestion`
- `set_congestion_options`
- `remove_congestion_options`
- `report_congestion_options`
- `report_congestion`
- `gui_show_map`

For command syntax and details, see the man pages.

Improving Area Correlation and Runtime with Synopsys Physical Guidance (SPG)

The physical guidance feature enables Design Compiler Graphical to save physical guidance information and pass this information to IC Compiler. With this information, IC Compiler can begin the implementation flow with the `place_opt` command. IC Compiler no longer needs to create a coarse placement by running commands such as `create_placement`, `remove_buffer_tree`, or `psynopt`. By using the Design Compiler physical guidance as a starting point for placement, runtime and area correlation with IC Compiler are improved.

This section describes the Synopsys Physical Guidance (SPG) feature in the following sections:

- Using Physical Guidance in Design Compiler (compile_ultra -spg)
- Using Physical Guidance in IC Compiler (place_opt -spg)
- Supported Flows
- Reporting Physical Guidance Information
- Physical Guidance Limitations

Using Physical Guidance in Design Compiler (compile_ultra -spg)

To enable physical guidance in Design Compiler, use the `-spg` option with the `compile_ultra` command.

Setup requirements:

Both Design Compiler and IC Compiler must use consistent physical and logical constraints and use the same libraries. The same DEF file must be used by both tools. The DEF file contains floorplan information and this information must be consistent between tools. The tools will issue various errors and warnings if your constraints, floorplan information, and libraries are not consistent.

Using Physical Guidance in IC Compiler (place_opt -spg)

To enable physical guidance in IC Compiler, use the `-spg` option with the `place_opt` command.

Setup requirements:

Both Design Compiler and IC Compiler must use consistent physical and logical constraints and use the same libraries. The same DEF file must be used by both tools. The DEF file contains floorplan information and this information must be consistent between tools. The tools will issue various errors and warnings if your constraints, floorplan information, and libraries are not consistent.

Supported Flows

The physical guidance feature supports all DC Ultra topographical mode compile options, such as `compile_ultra -spg -scan`, `compile_ultra -spg -scan -clock_gate`, `compile_ultra -spg -scan -congestion`, and `compile_ultra -spg -scan -incremental`.
In addition, the physical guidance feature supports the multicorner-multimode based flow, the multivoltage-UPF flow, the power flow, the DFT flow, and the hierarchical flow. Note that for the hierarchical flow, you cannot use ILMs that have been created by Design Compiler. You can use only IC Compiler ILMs. See "Physical Guidance Limitations" on page 10-93.

Reporting Physical Guidance Information

The physical guidance information is stored with the design in binary format. It is not available as a standalone file. To view the physical guidance information, you must run IC Compiler and execute the `place_opt -spg` command.

Physical Guidance Limitations

Design Compiler can only save the physical guidance information in .ddc file format or Milkyway format. The ASCII output netlist format is not supported.

In the hierarchical flow, only created ILMs that have been created by IC Compiler are supported. You cannot use ILMs created by Design Compiler or .ddc formatted ILMs.

Creating and Modifying Floorplans Using Floorplan Exploration

This section describes how to use floorplan exploration, which allows you to create and modify floorplans using Design Compiler Graphical.

This section contains the following subsections:

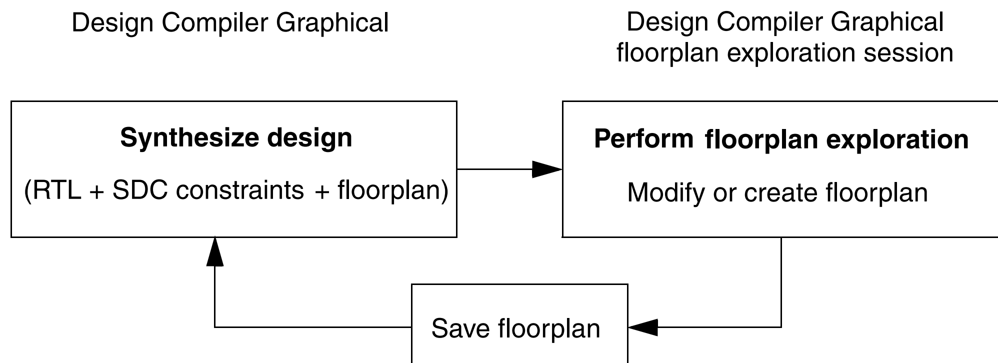
- Floorplan Exploration Overview
- Enabling Floorplan Exploration
- Using the Floorplan Exploration GUI
- Creating and Editing Floorplans
- Saving the Floorplan or Discarding Updates
- Saving the Floorplan into a Tcl Script File or DEF File
- Exiting the Session
- Incremental or Full Synthesis after Floorplan Changes
- Floorplan Exploration Limitations

Floorplan Exploration Overview

Floorplan exploration allows you to perform floorplanning tasks, including floorplan analysis, floorplan creation, and floorplan modification from within the synthesis environment. Design Compiler Graphical floorplan exploration uses the IC Compiler floorplanning tools in the IC Compiler layout window. Although you use the IC Compiler layout window, the interface between floorplan exploration in Design Compiler Graphical and the IC Compiler layout window is transparent, allowing you to move seamlessly between the Design Vision and IC Compiler layout windows.

Figure 10-24 shows a typical floorplan exploration flow. You read an RTL file, specify the physical and logic libraries, define the Synopsys design constraints, specify a floorplan (if you have one), and synthesize the design in Design Compiler Graphical. After synthesis, you evaluate the QoR results and use floorplan exploration to create a new floorplan or improve an existing floorplan, as needed.

Figure 10-24 Floorplan Exploration Flow



Enabling Floorplan Exploration

Before you begin floorplan exploration, you must have the following:

- A synthesized design

  Design Compiler Graphical issues an error if it finds any unmapped logic.

- Both the physical and logic libraries

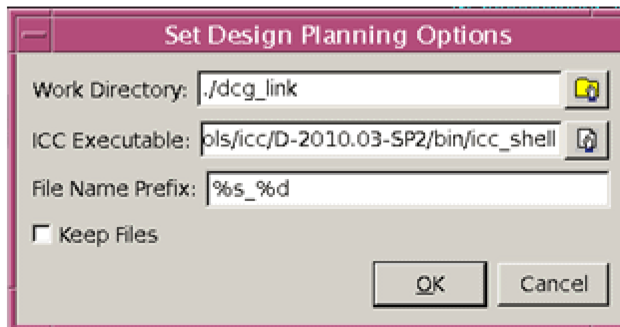
  All cells in the design must link to the physical and logic libraries. By default, Design Compiler Graphical creates a dummy physical cell if a cell is missing from the user-supplied physical libraries. However, in floorplan exploration, the tool issues an error and does not start the floorplan exploration session if your design is linked to a dummy physical cell.

Once you have a synthesized design that links to the physical and logic libraries, you must set the design planning options, as described in the following steps:

1. In the Design Compiler Graphical GUI, open the Design Vision layout window.
2. Choose Floorplan > Set Design Planning Options.

   The Set Design Planning Options dialog box appears, as shown in Figure 10-25.

Figure 10-25 Set Design Planning Options Dialog Box



3. In the Work Directory box, enter the name of your working directory with the relative path from the current directory.

   This is where the floorplanning scripts, the floorplan, and any other necessary files are stored.

4. In the ICC Executable box, enter the name and location of the IC Compiler executable.

   By default, Design Compiler uses the icc_shell executable specified by the $path variable.

5. In the File Name Prefix box, enter the prefix you want to use in the file name for any generated files, such as the floorplanning scripts and floorplan files.

   The default file prefix naming style is *%s_%d* where *%s* is the design name and *%d* is the process ID.

6. Select the Keep Files option if you want to retain the files that are created during and after the floorplan exploration session, such as the floorplanning scripts, floorplan files, and so on.

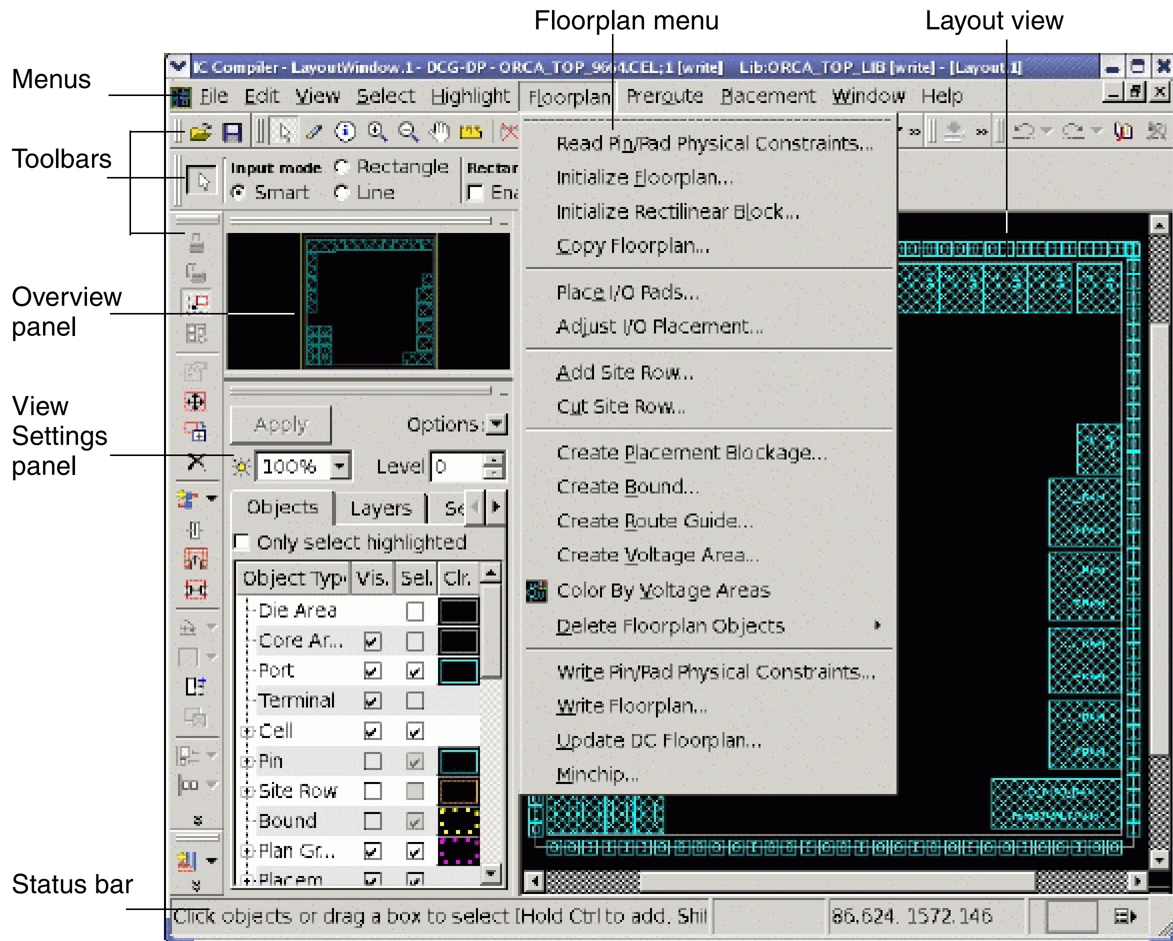
   This option is deselected by default, which means that the generated files are removed when the floorplan exploration session is closed.

7. Click OK.

Using the Floorplan Exploration GUI

Floorplan exploration in Design Compiler Graphical uses the IC Compiler layout window. By default, when you enable floorplan exploration, Design Compiler Graphical configures the IC Compiler layout window so that it includes only the menus and menu commands you need to perform Design Compiler Graphical floorplanning. Figure 10-26 shows the simplified floorplanning menu structure.

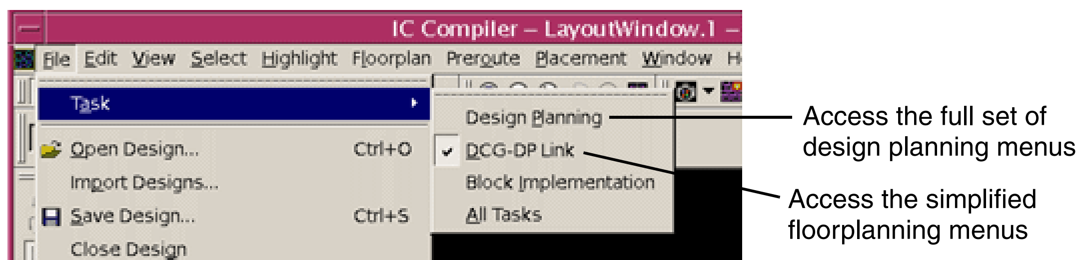
Figure 10-26 IC Compiler Simplified Layout Window Features



If you are an experienced IC Compiler user, you can use the full set of design planning menus.

You can access the full set of IC Compiler design planning menus by choosing File > Task > Design Planning, as shown in Figure 10-27. To switch from the full design planning menu structure back to the simplified menu structure, choose File > Task > DCG-DP Link.

Figure 10-27 Switching Between Simplified and Full Design Planning Menu Structures



Any work you complete using the full set of IC Compiler design planning menus is available when you switch back to the simplified floorplanning menus.

For general information about working in the IC Compiler layout window, see the IC Compiler Help system and the *IC Compiler Design Planning User Guide*.

Creating and Editing Floorplans

Whether you use the simplified Design Compiler Graphical floorplanning menus or the full set of IC Compiler design planning menus, the IC Compiler layout window provides interactive tools and commands that you can use to create or modify your floorplan. You can

- Use editing tools to move, resize, copy, split, reshape, or remove objects.
- Use editing commands to rotate, align, distribute, spread, or expand objects or to change cell orientations.

Shortcut keys are available for the most frequently performed editing operations. Online Help pages are provided for each of the editing tools.

In addition, you can use commands on the Floorplan menu to create physical objects such as placement blockages, bounds, and routing keepouts. When you create an object, you can specify its coordinates or draw the object in the layout view.

For more information about working with editing tools and commands in the IC Compiler layout window, see the IC Compiler Help system. For sample flows that show how to create a floorplan using Design Compiler Graphical floorplan exploration and how to modify a floorplan to improve timing or congestion, see the *Design Compiler Graphical Push-Button Floorplan Exploration Application Note* on Solvnet.

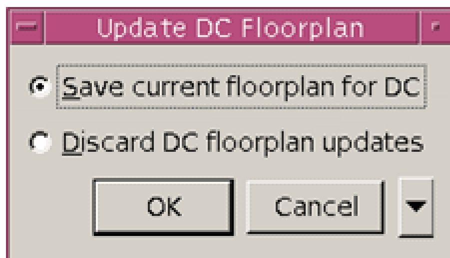
Saving the Floorplan or Discarding Updates

You can save or discard floorplan updates in Design Compiler Graphical from the IC Compiler floorplanning session at any time using the “Update DC Floorplan” command. To save or discard the floorplan, perform the following steps:

1. Choose Floorplan > Update DC Floorplan.

   The Update DC Floorplan dialog box appears, as shown in Figure 10-28.

Figure 10-28 Update DC Floorplan Dialog Box



   The “Update DC Floorplan” command is available only in the simplified floorplanning menu structure for Design Compiler Graphical. For information about switching from the full set of design planning menus to the simplified floorplanning menu structure, see “Using the Floorplan Exploration GUI” on page 10-97.

2. Choose one of the following options:

   - To save the current floorplan in Design Compiler Graphical, select the “Save current floorplan for DC” option.

     If you have previously saved a floorplan, the tool overwrites it.

   - To remove a previously saved floorplan, select the “Discard DC floorplan updates” option.

3. Click OK.

Saving the Floorplan into a Tcl Script File or DEF File

You can also save the floorplan into a Tcl script file or a DEF file during the floorplan exploration session. However, the Tcl or DEF file is not automatically read in Design Compiler Graphical.

To save the floorplan into a Tcl script file, choose Floorplan > Write Floorplan, and specify the required options in the Write Floorplan dialog box.

Alternatively, you can use the `write_floorplan` command with its required options at the tool prompt.

To save the floorplan in a DEF file, choose File > Export > Write DEF, and specify the required options in the Write DEF dialog box.

Alternatively, you can use the `write_def` command with its required options at the tool prompt.

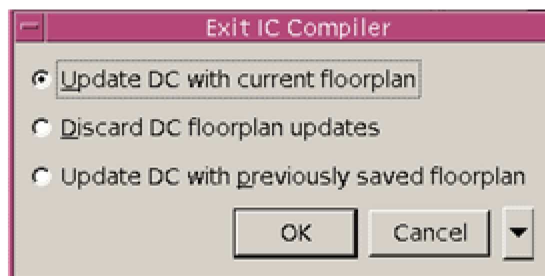
Exiting the Session

To exit the floorplan exploration session and save your floorplan for synthesis or discard the floorplan, perform the following steps:

1. Choose File > Exit in the menu.

   The Exit IC Compiler dialog box appears, as shown in Figure 10-29.

Figure 10-29 Exit IC Compiler Dialog Box



2. Choose one of the following options:

   - Select the “Update DC with current floorplan” option to save the current floorplan in Design Compiler Graphical and exit the floorplan exploration session.
   - To exit the floorplan exploration session and discard the floorplan, including any previously saved floorplans, select the “Discard DC floorplan updates” option.
   - To keep the floorplan that you saved previously in Design Compiler Graphical, discard any subsequent changes that you have not saved, and exit the floorplan exploration session, select the “Update DC with previously saved floorplan” option.

3. Click OK.

When you exit the floorplan exploration session, the Design Compiler Graphical windows and shell interface become active.

If you update the floorplan, the tool closes the original Design Vision layout window. To see the floorplan changes, open a new Design Vision layout window.

Incremental or Full Synthesis after Floorplan Changes

The floorplan changes impact the design only through synthesis. After you create a floorplan or modify an existing floorplan, you can perform incremental synthesis by using the `compile_ultra -incremental` command for a quick assessment of the floorplan's QoR benefits. However, incremental synthesis might not meet the requirements for achieving good QoR results or the requirements for a final implementation flow. In these cases, perform full synthesis using the `compile_ultra` command.

To prepare for full synthesis, you must save the floorplan into a dctcl script file by running the `write_floorplan` command at the tool prompt, as shown:

```
write_floorplan -all DESIGN.fp
```

The `-all` option is recommended for resynthesizing the design with the new or modified floorplan.

Use the `read_floorplan` command to read the floorplan in Design Compiler for synthesis. After you read the RTL file, the SDC constraints, and the new floorplan file, you perform full synthesis with the `compile_ultra` command.

For information about the `read_floorplan` and `write_floorplan` commands, see "Using Floorplan Physical Constraints" on page 10-15. For information about synthesizing the design, see "Overview of Topographical Technology" on page 10-3.

Floorplan Exploration Limitations

Floorplan exploration in Design Compiler Graphical has the following limitations:

- Design Compiler topographical mode does not support keepout margins, but it recognizes placement blockages. For information about converting keepout margins to placement blockages, see the *Design Compiler Graphical Push-Button Floorplan Exploration Application Note*.
- Design Compiler ILMs (wire load or topographical) and Design Compiler physical hierarchies are not supported in IC Compiler.
- Some floorplan constraints are transformed when read in Design Compiler Graphical. For example, vias are converted to net shapes.
- Some floorplan constraints, such as tracks and keepout margins, are ignored when read in Design Compiler Graphical.
- Relative placement constraints are not passed to and from the IC Compiler floorplanning session.

Optimizing Multicorner-Multimode Designs in Design Compiler Graphical

Multicorner-multimode design optimization is a feature available only in the Design Compiler Graphical tool. This feature enables you to analyze and optimize designs across multiple modes and multiple corners concurrently. For more details about other Design Compiler Graphical features, see "Using the Design Compiler Graphical Tool" on page 10-84.

This section describes multicorner-multimode design optimization, in the following subsections:

- Multicorner-Multimode Concepts
- Multicorner-Multimode Feature Support
- Unsupported Features
- Concurrent Multicorner-Multimode Optimization and Timing Analysis
- Basic Multicorner-Multimode Flow
- Setting Up the Design for a Multicorner-Multimode Flow
- Handling Libraries in the Multicorner-Multimode Flow
- Scenario Management Commands
- Reporting Commands
- Supported SDC Commands
- Multicorner-Multimode Script Example
- Using ILMs in Multicorner-Multimode Designs

Multicorner-Multimode Concepts

Designs must often operate under multiple operating conditions, often called corners, and in multiple modes. Such designs are referred to as multicorner-multimode designs. Design Compiler Graphical extends the topographical technology to analyze and optimize these designs across multiple modes and multiple corners concurrently. The multicorner-multimode feature also provides ease-of-use and compatibility between flows in Design Compiler and IC Compiler due to the similar user interface. Common multicorner-multimode terms are described below.

Terminology

- Corner

  A *corner* is defined as a set of libraries characterized for process, voltage and temperature (PVT) variations. Corners are not dependent on functional settings; they are meant to capture variations in the manufacturing process, along with expected variations in the voltage and temperature of the environment in which the chip will operate.

- Mode

  A *mode* is defined by a set of clocks, supply voltages, timing constraints, and libraries. It can also have annotation data, such as SDF or parasitics files. Multicorner-multimode designs can operate in many modes such as the test mode, mission mode, standby mode and so forth.

- Scenario

  A scenario is a combination of modal constraints and corner specifications. In a design, the tool uses a scenario or a set of scenarios as the unit for multimode-multicorner analysis and optimization. Some constraints can be part of both the mode and corner specification. Optimization of multicorner-multimode design involves managing the scenarios of the design. For more details on scenario management, see “Scenario Management Commands” on page 10-114.

Basic Scenario Definition

To define modes and corners, you use the `create_scenario` command. A scenario definition includes commands that specify the TLUPlus libraries, operating conditions, and constraints, as shown in Example 10-42, which defines the s1 scenario. A scenario definition must include the `set_operating_conditions` and `set_tlu_plus_files` commands. The following sections describe these commands along with the associated library setup information that is needed to run multicorner-multimode design optimization.

Example 10-42 Basic Scenario Definition

```
create_scenario s1
set_operating_conditions WORST -library stdcell.setup.typ.db:stdcell_typ
set_tlu_plus_files -max_tluplus design.tlup -tech2itf_map layermap.txt
read_sdc s1.sdc
```

Multicorner-Multimode Feature Support

The multicorner-multimode feature in Design Compiler Graphical provides compatibility between flows in Design Compiler and IC Compiler through the use of a similar user interface.

Keep the following points in mind when running multicorner-multimode optimizations:

- All options of the `compile_ultra` command are supported.
- Multicorner-multimode technology is supported only in topographical mode with the DC-Extension license. Multicorner-multimode technology is not supported in DC Expert.
- Leakage power optimizations are supported.
- Dynamic power optimizations are supported.
- The UPF flow is supported only for multivoltage designs.

Unsupported Features

The following features are not supported in Design Compiler Graphical for multicorner-multimode designs:

- Power-driven clock gating is not supported.

  However, if you use the `compile_ultra -gate_clock` or the `insert_clock_gating` commands, clock-gate insertion is performed on the design, independent of the scenarios.
- Clock tree estimation is not supported.
- k-factor scaling is not supported.

  Because multicorner-multimode design libraries do not support the use of k-factor scaling, the operating conditions that you specify for each scenario must match the nominal operating conditions of one of the libraries in the list of the link libraries.
- The `set_min_library` command is not supported for individual scenarios.

  The `set_min_library` command applies to all scenarios. If you use `set_min_library` to define one scenario, the tool will use the library for all scenarios.

Concurrent Multicorner-Multimode Optimization and Timing Analysis

Concurrent multicorner-multimode optimization works on the worst violations across all scenarios, eliminating the convergence problems observed in sequential approaches.

Timing analysis is carried out on all scenarios concurrently, and costing is measured across all scenarios for timing and design rules. As a result, the timing and constraint reports show worst-case timing across all scenarios.

To run timing analysis, use one of the following two methods:

- Traditional minimum-maximum analysis

  To use this analysis method, define your analysis type as `bc_wc`, for example,

  ```
  set_operating_conditions -analysis_type bc_wc
  ```

- Early-late analysis

  To use this analysis method, define your analysis type as `on_chip_variation`, for example,

  ```
  set_operating_conditions -analysis_type on_chip_variation
  ```

  PrimeTime also uses the on-chip variation (OCV) method.

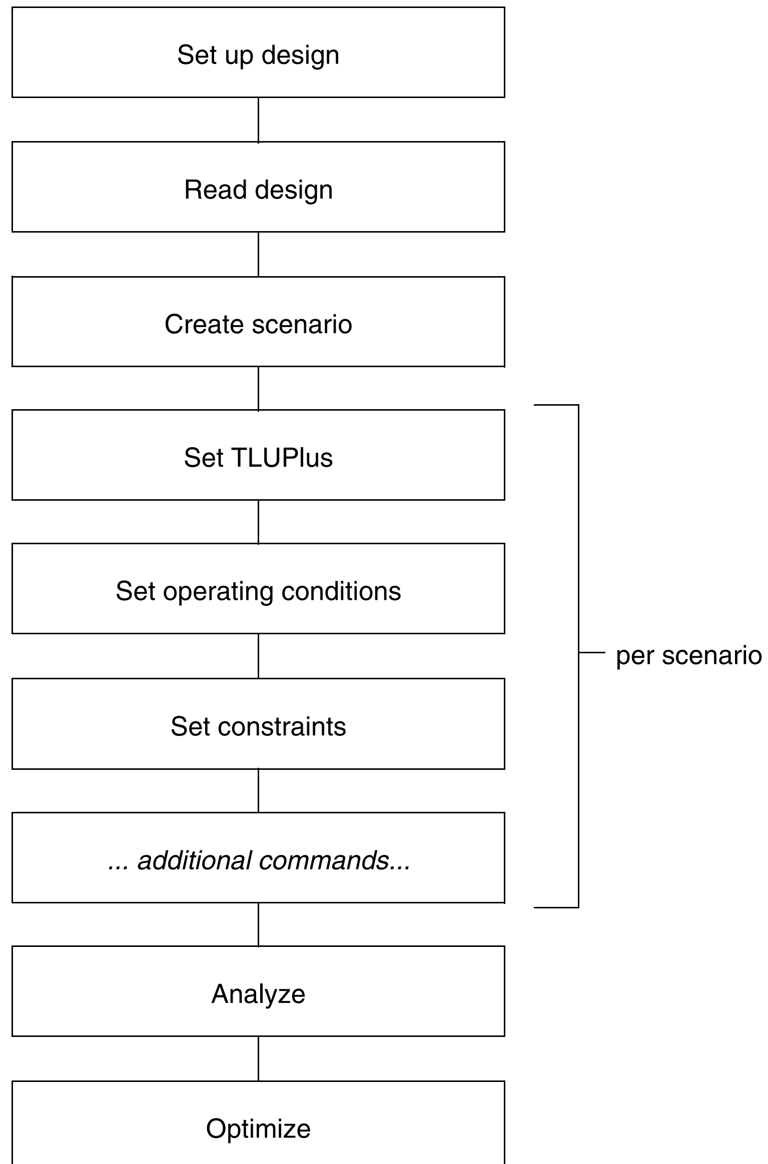
Basic Multicorner-Multimode Flow

Figure 10-30 shows the basic multicorner-multimode flow.The key step in multicorner-multimode optimization involves creating the scenarios. You use the `create_scenario` command to create the scenarios. You can create multiple scenarios. For each scenario, you set constraints specific to the scenario mode and you set operating conditions specific to the scenario corner. For an example script, see “Multicorner-Multimode Script Example” on page 10-129.

Scenario definitions include commands that specify the TLUPlus libraries, operating conditions, and constraints. However, other commands can be included. For example, you can use the `set_max_leakage_power` command to control leakage power on a per-scenario basis or you can use the `read_sdf` command to set the correct net RC and pin-to-pin delay information in the respective scenarios.

After you configure all the scenarios, you can activate a subset of these scenarios by using the `set_active_scenarios` command.

Figure 10-30 Basic Multimode Flow



Setting Up the Design for a Multicorner-Multimode Flow

To setup a design for a multicorner-multimode flow, you must specify the TLUPlus files, operating conditions, and Synopsys Design Constraints for each scenario. Design Compiler uses the nominal process, voltage, and temperature (PVT) values to group the libraries into different sets. Libraries with the same PVT values are grouped into the same set. For each scenario, the PVT of the maximum operating condition is used to select the appropriate set. Setup considerations are described in the following sections:

- Specifying TLUPlus Files
- Specifying Operating Conditions
- Specifying Constraints

Specifying TLUPlus Files

Use the `set_tlu_plus_files` command to specify the TLUPlus files for each scenario, as shown in Example 10-43.

Example 10-43 Specifying TLUPlus Files for a Scenario

```
create_scenario s1
set_operating_conditions WORST -library stdcell.setup.typ.db:stdcell_typ
set_tlu_plus_files -max_tluplus design.tlup -tech2itf_map layermap.txt
read_sdc s1.sdc
```

If you do not specify a TLUPlus file, the tool returns an error message similar to the following:

```
Error: tlu_plus files are not set in this scenario s1.
      RC values will be 0.
```

If a TLUPlus library is not correct, the tool issues the following error message:

```
Error: TLU+ sanity check failed (OPT-1429)
```

If you want to enable temperature scaling, the TLUPlus files must contain the `GLOBAL_TEMPERATURE` and `CRT1` variables. The `CRT2` variable is optional. The following example is an excerpt from a TLUPlus file:

```
TECHNOLOGY = 90nm_lib
GLOBAL_TEMPERATURE = 105.0
CONDUCTOR metal8 {THICKNESS= 0.8000
   CRT1=4.39e-3 CRT2=4.39e-7
...
```

Specifying Operating Conditions

You must define the operating condition for each scenario. You specify different operating conditions for different scenarios using the `set_operating_condition` command, as shown in Example 10-44.

Example 10-44 Specifying Operating Conditions for a Scenario

```
create_scenario s1
set_operating_conditions SLOW_95 -library max_vmax_v95_t125
set_tlu_plus_files -max_tluplus design.tlup -tech2itf_map layermap.txt
read_sdc s1.sdc
```

If you do not define an operating condition for a scenario, the tool issues MV-020 and MV-021 warnings.

Specifying Constraints

For each scenario, you must specify the Synopsys Design Constraints (SDCs) specific to that scenario, as shown in Example 10-45.

Example 10-45 Specifying SDC Constraints for a Scenario

```
create_scenario s1
set_operating_conditions WORST -library stdcell.setup.typ.db:stdcell_typ
set_tlu_plus_files -max_tluplus design.tlup -tech2itf_map layermap.txt
read_sdc s1.sdc
```

The tool discards any previous scenario-specific constraints after you execute the `create_scenario` command and reports an MV-020 warning, as shown in Example 10-46.

Example 10-46 Tool Removes Previous Constraints

```
dc_shell-topo> create_scenario s1
Warning: Any existing scenario-specific constraints
      are discarded. (MV-020)
dc_shell-topo> report_timing
Warning: No operating condition was set in scenario s1 (MV-021)
```

Handling Libraries in the Multicorner-Multimode Flow

The following sections discuss how to handle libraries in multicorner-multimode designs:

- Using Link Libraries That Have the Same PVT Nominal Values
- Using Unique PVT Names to Prevent Linking Problems
- Unsupported k-factors
- Automatic Detection of Driving Cell Library
- Defining Minimum Libraries

Using Link Libraries That Have the Same PVT Nominal Values

The link library lists all the libraries that are to be used for linking the design for all scenarios. Furthermore, because several libraries are often intended for use with a particular scenario, such as a standard cell library and a macro library, Design Compiler automatically groups the libraries in the link library list into sets and identifies which set must be linked with each scenario.

The tool groups libraries according to the PVT value of the library. Libraries with the same PVT values are grouped into the same set. The tool uses the PVT value of a scenario's maximum operating condition to select the appropriate set for the scenario.

If the tool finds no suitable cell in any of the specified libraries, an error is reported as shown in the following example,

```
Error: cell TEST_BUF2En_BUF1/Z (inx4) is not characterized
      for 0.950000V, process 1.000000,
      temperature -40.000000. (MV-001)
```

You should verify the operating conditions and library setup. You must fix this error before you can optimize your design.

Link Library Example

To understand how library linking works, consider Table 10-12.

Table 10-12 shows the libraries in the link library list, their nominal PVT values, and the operating condition, if any, that is specified in each library. The design has instances of combinational, sequential, and macro cells.

Table 10-12 Link Libraries With PVT and Operating Conditions

Link library (in order)	Nominal PVT	Operating conditions in library (PVT)
Combo_cells_slow.db	1/0.85/130	WORST (1/0.85/130)
Sequentials_fast.db	1/1.30/100	None
Macros_fast.db	1/1.30/100	None
Macros_slow.db	1/0.85/130	None
Combo_cells_fast.db	1/1.30/100	BEST (1/1.3/100)
Sequentials_slow.db	1/0.85/130	None

To create the scenario, s1, with cell instances linked to the Combo_cells_slow, Macros_slow, and Sequential_slow libraries, you run

```
dc_shell-topo> create_scenario s1
dc_shell-topo> set_operating_conditions -max WORST -library slow
```

where the library “slow” is defined within the Combo_cells_slow.db file. Note that using the `-library` option in the `set_operating_conditions` command helps the tool identify the correct PVT for the operating conditions. The PVT of the maximum operating condition is used to find the correct matches in the link library list during linking.

Using this library linking method, you can link libraries that do not have operating condition definitions. The method also enables you to have multiple library files. For example, you can have one library file for standard cells, another for macros, and so forth.

Inconsistent Libraries Warning

When you use multiple libraries, if library cells with the same name are not functionally identical or do not have identical sets of library pins with the same name and order, the tool issues a warning stating that the libraries are inconsistent.

You should run the `check_library` command before running a multicorner-multimode flow, as shown in the following example,

```
set_check_library_options -mcmm
check_library -logic_library_name {a.db b.db}
```

When you use the -mcmm option with the set_check_library_options command, the check_library command performs multicorner-multimode specific checks, such as determining operating condition or power-down inconsistencies. When inconsistencies are detected, the tool generates a report that lists the inconsistencies. In addition, the tool issues the following summary information message:

```
Information: Logic library consistency check FAILED for MCMM.
(LIBCHK-360)
```

When you get a LIBCHK-360 message, check the report to identify the cause of the problem and fix the library inconsistencies. The LIBCHK-360 man page describes possible causes for the library inconsistencies.

Setting the dont_use Attribute on Library Cells in the Multicorner-Multimode Flow

When you set the dont_use attribute on a library cell, the multicorner-multimode feature requires that all characterizations of this cell have the dont_use attribute. Otherwise, the tool might consider the libraries as inconsistent. You can use the wildcard character to set the dont_use attribute as follows:

```
set_dont_use */AN2
```

Note that you do not have to issue the command multiple times to set the dont_use attribute on all characterizations of a library cell.

Using Unique PVT Names to Prevent Linking Problems

To prevent linking problems, make sure your PVT operating conditions have unique names. If the maximum libraries associated with each scenario do not have distinct PVT values, your cell instances might be incorrectly linked, resulting in incorrect timing values. This happens because the nominal PVT values that are used to group the link libraries into sets group the maximum libraries of different corners into one set. Consequently, the cell instances are linked to the first cell with a matching type in that set, for example, the first AND2_4 cell, even though the -library option is specified for each of the scenario-specific set_operating_conditions commands. That is, the -library option locates the operating condition and its PVT but not the library to link.

The following paragraphs describe a linking problem due to non-unique PVT names. For the conditions given in Table 10-13, Table 10-14, and Example 10-47, the tool groups the Ftyp.db and TypHV.db libraries into a set with Ftyp.db as the first library in the set. Therefore, the cell instances in scenario s2 are not linked to the library cells in TypHV.db, as intended. Instead, they are incorrectly linked to the library cells in the Ftyp.db library, assuming that all the libraries include the library cells required to link the design.

Table 10-13 shows the libraries in the link library, listed *in order*; their nominal PVT; and the operating condition that is specified in each library.

Table 10-13 Link Libraries With PVT and Operating Conditions

Link library (in order)	**Nominal PVT**	**Operating conditions in library (PVT)**
Ftyp.db	1/1.30/100	WORST (1/1.30/100)
Typ.db	1/0.85/100	WORST (1/0.85/100)
TypHV.db	1/1.30/100	WORST (1/1.30/100)
Holdtyp.db	1/0.85/100	BEST (1/0.85/100)

Table 10-14 shows the operating condition specifications for each of the scenarios; Example 10-47 shows the corresponding scenario creation script.

Table 10-14 Scenarios and Their Operating Conditions

	Scenarios			
	s1	**s2**	**s3**	**s4**
Max Opcond (Library)	WORST (Typ.db)	WORST (TypHV.db)	WORST (Ftyp.db)	WORST (Typ.db)
Min Opcond (Library)	None	None	None	BEST (HoldTyp.db)

Example 10-47 Linking Problem Due to Non-Unique PVT Name

```
create_scenario s1
set_operating_conditions WORST -library Typ.db:Typ
create_scenario s2
set_operating_conditions WORST -library TypHV.db:TypHV
create_scenario s3
set_operating_conditions WORST -library Ftyp.db:Ftyp
create_scenario s4
set_operating_condition \
     -max WORST -max_library Typ.db:Typ \
     -min BEST -min_library HoldTyp.db:HoldTyp
```

Ambiguous Libraries Warning

The tool issues a warning if your design uses any libraries containing cells with the same name and same nominal PVT. The warning states that the libraries are ambiguous and identifies which libraries are being used and which are being ignored.

Unsupported k-factors

Multicorner-multimode design libraries do not support k-factor scaling. Therefore, the operating conditions that you specify for each scenario must match the nominal operating conditions of one of the libraries in the link library list.

Automatic Detection of Driving Cell Library

In multicorner-multimode flow, the operating condition setting is different for different scenarios. To build the timing arc for the driving cell, different technology libraries are used for different scenarios. You can specify the library using the `-library` option of the `set_driving_cell` command. But specifying the library is optional because the tool can automatically detect the driving cell library.

When you specify the library using the `-library` option of the `set_driving_cell` command, the tool searches for the specified library in the link library set. If the specified library exists, it is used. If the specified library does not exist in the link library, the tool issues the UID-993 error message as follows:

```
Error: Cannot find the specified driving cell in memory.(UID-993)
```

When you do not use the `-library` option of the `set_driving_cell` command, the tool searches all the libraries for the matching operating conditions. The first library in the link library set that matches the operating condition is used. If no library in the link library set matches the operating condition, the first library in the link library set that contains the matching library cell is used. If no library in the link library set contains the matching library cell, the tool issues the UID-993 error message.

Defining Minimum Libraries

Minimum libraries are usually defined with the `set_operating_conditions` command. You can use the `set_min_library` command, but it is not scenario-specific. If you use `set_min_library` to define a minimum library for a scenario, the tool uses that library as the

minimum library for all scenarios, even if you do not define that library in all your scenarios. If you want to define different minimum libraries for each scenario, use the `set_operating_conditions` command.

Table 10-15 Unsupported Multiple Minimum Library Configuration

	Scenarios	
	s1	**s2**
Max library	Slow.db	Slow.db
Min library	Fast_0yr.db	Fast_10yr.db

For example, you could not relate two different minimum libraries – say, Fast_0yr.db and Fast_10yr.db – with the maximum library, Slow.db, in two separate scenarios. The first minimum library you specify would apply to both scenarios. Table 10-15 shows the *unsupported* configuration.

Note, however, that a minimum library can be associated with multiple maximum libraries. As shown in Table 10-16, the minimum library Fast_0yr.db is paired with both the maximum library Slow.db of scenario 1 and the maximum library SlowHV.db of scenario 2.

Table 10-16 Supported Min-Max Library Configuration

	Scenarios	
	s1	**s2**
Max library	Slow.db	SlowHV.db
Min library	Fast_0yr.db	Fast_0yr.db

Scenario Management Commands

Use the following commands to create and manage scenarios:

- `create_scenario`
- `current_scenario`
- `all_scenarios`
- `all_active_scenarios`

- `set_active_scenarios`
- `set_scenario_options`
- `set_preferred_scenarios`
- `check_scenarios`
- `remove_scenario`
- `report_scenarios`
- `report_scenario_options`

The following sub-sections describe how you use these commands to manage scenarios:

- Creating Scenarios
- Defining Active Scenarios
- Scenario Reduction
- Specifying Scenario Options
- Removing Scenarios

Creating Scenarios

You use the `create_scenario` command to create a new scenario. When the first scenario is created, all previous scenario-specific constraints are removed from the design and the following warning is issued:

```
dc_shell-topo> create_scenario s1
Warning: Any existing scenario-specific constraints
      are discarded. (MV-020)
Current scenario is: s1
```

Use the `current_scenario` command to specify the name of the current scenario. Without arguments, the command returns the name of the current scenario. Note that the `current_scenario` command merely specifies the focus scenario and that when you define more than one scenario, the tool performs concurrent analysis and optimization across all scenarios, independent of the current scenario setting.

Use the `set_tlu_plus` command to set the TLUPlus files in the scenario specified by the `current_scenario` command. Note that each scenario must have a set of TLUPlus files specified, or the following error is reported:

```
Error: tlu_plus files are not set in this scenario <name>.
RC values will be 0.
```

Defining Active Scenarios

During concurrent analysis and optimization, you can significantly reduce memory usage and runtime by limiting the number of active scenarios to those that are “essential” or “dominant.” An essential or dominant scenario has the worst slack among all the scenarios for at least one of its constrained objects. (Constrained objects can include delay constraints associated with a pin, the design rule constraints for a net, leakage power, and so on.) Any scenario for which one of its constrained objects is the worst slack value is a dominant scenario.

You use the `set_active_scenarios` command to define active scenarios. Other commands related to active scenarios are `all_scenarios` and `all_active_scenarios`.

Scenario Reduction

Topographical mode automatically performs scenario reduction on the current set of active scenarios to reduce memory and runtime. In general, restricting concurrent analysis and optimization to a subset of dominant scenarios does not lead to a significant difference in QoR. The tool analyzes all the active scenarios and automatically determines a set of dominant scenarios, based on the number of violating endpoints, and optimizes them for timing, power, and design rule. In the current version, topographical mode does not support the `get_dominant_scenario` command, which IC Compiler supports. However, you can choose a preferred scenario by using the `set_preferred_scenario` command. When you set the preferred scenario, the tool treats it as the most constraining scenario. It still performs scenario reduction to determine the dominant scenarios but treats the user-specified preferred scenario as the most constraining one.

Specifying Scenario Options

Use the `set_scenarios_options` command to define specific constraint options, such as leakage power, that you want optimized in a scenario. You can apply the constraint option to more than one scenario at a time by using the `-scenarios` option. The options can be specified on both active and inactive scenarios. If you do not specify any scenario, the options are applied only to the current scenario.

Use the `-leakage_power` option to enable or disable the leakage power optimization. The default value of the `-leakage_power` option is `true`. Leakage power optimization is enabled by default on all scenarios. Set the `-leakage_power` option to `false` to disable the leakage power optimization. The following command shows how to enable leakage power optimization on only specific scenarios.

```
set_scenario_options -leakage_power true \
          -scenarios scenario_list
```

Use the `-dynamic_power` option to enable or disable the dynamic power optimization. The default value of the `-dynamic_power` option is `true`. Dynamic power optimization is enabled by default on all scenarios.

Use the `-setup` option to enable or disable the setup or maximum delay optimization for specified scenarios. The default value of the `-setup` option is `true`, so maximum delay optimization is enabled by default. The following example shows how to disable maximum delay optimization on specific scenarios.

```
dc_shell-topo> set_scenario_options -setup false -scenarios
scenario_list
```

Use the `-hold` option to enable or disable the hold or minimum delay optimization for the specified scenarios. The default value of the `-hold` option is `true`. When you set the `-hold` option to `false`, the tool ignores the hold or the minimum delay violations in the specified scenarios.

```
dc_shell-topo> set_scenario_options -reset_all true \
          -scenarios [all_scenarios]
```

Use the `report_scenario_options` command to report the scenario options.

Removing Scenarios

Use the `remove_scenario` command to remove the specified scenarios. All scenario-specific constraints defined in the removed scenario are deleted. For example,

```
dc_shell-topo> remove_scenarios
s1 s2
dc_shell-topo> remove_scenario -all
Removed scenario 's2'
Removed scenario 's1'
dc_shell-topo> all_scenarios
```

Power Optimization Techniques

Design Compiler Graphical supports power optimization for multicorner-multimode designs. You use the `set_max_leakage_power` and `set_max_dynamic_power` commands to set the leakage and dynamic power constraints on specific scenarios of a multicorner-multimode design.

The following sections describe how you perform different types of leakage and dynamic power optimization on multicorner-multimode designs.

For details on multivoltage, multicorner-multimode design optimization, and the UPF flow for the multivoltage, multicorner-multimode design optimization, see the *Power Compiler User Guide*.

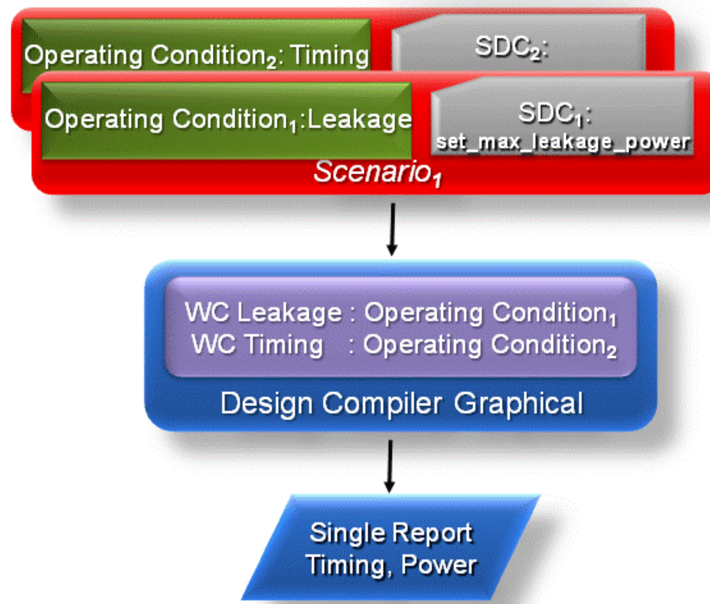
Optimizing for Leakage Power

Figure 10-31 shows how you set various constraints on different scenarios of a multicorner-multimode design.

Typically, in a multicorner-multimode design, leakage power optimization and timing optimization are done on different corners. Therefore, the worst case leakage corner can be different from a worst case timing corner. To perform leakage power optimization on specific corners, set the leakage power constraint on specific scenarios of the multicorner-multimode design by using the `set_max_leakage_power` command.

Note:
The `set_scenario_options` and `get_dominant_scenarios` commands are not supported in Design Compiler Graphical.

Figure 10-31 Setting Different Constraints on Different Scenarios



Note the following points when you optimize for leakage power in multicorner-multimode designs:

- You must define the leakage power constraint on specific scenarios targeted for leakage power optimization.
- Leakage and timing optimizations can be performed concurrently across multiple scenarios.
- The worst case leakage corner is different from the worst case timing corner.

The following example script shows how to create a scenario and set the leakage power constraint on the scenario:

Example 10-48 Performing Leakage Power Optimization in a Multicorner-Multimode Flow

```
read_verilog top.v
current_design top
link

create_scenario s1
set_operating_conditions WCCOM -library slow.db:slow
set_tlu_plus_files -max_tluplus max.tlu_plus -tech2itf_map tech.map
read_sdc ./s1.sdc
set_switching_activity -toggle_rate 0.25 -clock p_Clk -static_probability
0.015 -select inputs
set_max_leakage_power 0

create_scenario s2
set_operating_conditions BCCOM -library fast.db:fast
set_tlu_plus_files -max_tluplus max.tlu_plus -tech2itf_map tech.map
read_sdc ./s2.sdc

create_scenario s3
set_operating_conditions TCCOM -library typ.db:typ
set_tlu_plus_files -max_tluplus max.tlu_plus -tech2itf_map tech.map
read_sdc ./s3.sdc

create_scenario s4
set_operating_conditions NCCOM -library typ2.db:typ2
set_tlu_plus_files -max_tluplus max.tlu_plus -tech2itf_map tech.map
read_sdc ./s4.sdc
set_max_leakage_power 0

report_scenarios
compile_ultra -scan -gate_clock
report_power -scenario [all_scenarios]
report_timing -scenario [all_scenarios]
report_scenarios
report_qor
report_saif
```

Optimizing for Dynamic Power

To perform dynamic power optimization for a multicorner-multimode design, you must specify the `set_max_dynamic_power` command on every scenario of the design. Do not specify the dynamic power constraint only on certain scenarios. If you do so, Power Compiler issues an error message.

Note:
Unlike leakage power optimization where you specify the leakage constraint on specific scenarios, for dynamic power optimization, the dynamic power constraint must be specified on every scenario of the multicorner-multimode design.

The following example script shows how you set dynamic power constraints on the scenarios of a multicorner-multimode design.

Example 10-49 Performing Dynamic Power Optimization in a Multicorner-Multimode Design

```
create_scenario s1
read_sdc s1.sdc
set_operating_conditions WCCOM -library test1.db:test1
set_tlu_plus_files -max_tluplus max.tlu_plus -tech2itf_map tech.map
set_max_dynamic_power 0

create_scenario s2
read_sdc s2.sdc
set_operating_conditions BCCOM -library test2.db:test2
set_tlu_plus_files -max_tluplus max.tlu_plus -tech2itf_map tech.map
set_max_leakage_power 0
set_max_dynamic_power 0

create_scenario s3
read_sdc s3.sdc
set_operating_conditions NCCOM -library test3.db:test3
set_tlu_plus_files -max_tluplus max.tlu_plus -tech2itf_map tech.map
set_max_dynamic_power 0

create_scenario s4
read_sdc s4.sdc
set_operating_conditions WCCOM -library test4.db:test4
set_tlu_plus_files -max_tluplus max.tlu_plus -tech2itf_map tech.map
set_max_dynamic_power 0
```

Reporting Commands

This section describes the commands that you can use for reporting multicorner-multimode designs in the following sub-sections:

- report_scenario Command
- report_scenario_options Command
- Reporting Commands That Support the -scenario Option
- Commands That Report the Current Scenario
- Reporting Examples

report_scenario Command

The `report_scenario` command reports the scenario setup information for multicorner-multimode designs. The scenario specific information includes the technology library used, the operating condition, and TLUPlus files.

The following example shows a report generated by the `report_scenarios` command:

```
***************************************
Report : scenarios
Design : DESIGN1
scenario(s) : SCN1
Version: C-2009.06
Date   : Fri Apr 17 20:55:59 2009
***************************************

All scenarios (Total=4): SCN1 SCN2 SCN3 SCN4
All Active scenarios (Total=1): SCN1
Current scenario      : SCN1

Scenario #0: SCN1 is active.
Scenario options:
Has timing derate: No
Library(s) Used:
 technology library name (File: library.db)

Operating condition(s) Used:
  Analysis Type     : bc_wc
  Max Operating Condition: library:WCCOM
  Max Process     : 1.00
  Max Voltage     : 1.08
  Max Temperature: 125.00
  Min Operating Condition: library:BCCOM
  Min Process     : 1.00
  Min Voltage     : 1.32
  Min Temperature: 0.00

Tlu Plus Files Used:
  Max TLU+ file: tlu_plus_file.tf
  Tech2ITF mapping file: tf2itf.map
```

report_scenario_options Command

Use the `report_scenario_options` command to report the scenario options set by the `set_scenario_options` command. You can specify a list of scenarios to be reported by using the `-scenarios` option. By default, this command reports scenario options for the current, active scenario.

To illustrate the report generated by the `report_scenario_options` command, consider Example 10-50 where the `set_scenario_options -leakage_power false` command is executed. This command sets the `-leakage_power` option to `false`. With the `-leakage_power` option set to `false`, the tool will not optimize the current, active scenario for leakage power. This status is shown by the `report_scenario_options` report in Example 10-50.

Example 10-50

```
dc_shell-topo> set_scenario_options -leakage_power false
dc_shell-topo> report_scenario_options
****************************************
Report : scenario options
Design : TEST03
Version: D-2010.03
Date   : Wed Jan 20 14:57:08 2010
****************************************

Scenario: MODE1 is active.

setup         : true
hold          : true
leakage_power : false
dynamic_power : true
```

Reporting Commands That Support the -scenario Option

Some reporting commands support the `-scenario` option to report scenario-specific information. You can specify a list of scenarios to the `-scenario` option, and the tool reports scenario details for the specified scenarios.

The following reporting commands support the `-scenario` option:

- `report_timing`
- `report_timing_derate`
- `report_power`
- `report_clock`
- `report_path_group`
- `report_extraction_options`
- `report_tlu_plus_files`
- `report_constraint`

Commands That Report the Current Scenario

The following reporting commands report scenario-specific details for the current scenario. The header section of the report contains the name of the current scenario. No additional options are required to report the scenario-specific details of the current scenario.

- `report_net`
- `report_annotated_check`
- `report_annotated_transition`
- `report_annotated_delay`
- `report_attribute`
- `report_case_analysis`
- `report_ideal_network`
- `report_internal_loads`
- `report_clock_gating_check`
- `report_clock_tree`
- `report_clock_tree_power`
- `report_delay_calculation`
- `report_delay_estimate_options`
- `report_transitive_fanout`
- `report_disable_timing`
- `report_latency_adjustment_options`
- `report_net`
- `report_power_calculation`
- `report_noise`
- `report_signal_em`
- `report_timing_derate`
- `report_timing_requirements`
- `report_transitive_fanin`
- `report_crpr`
- `report_clock_timing`

Reporting Examples

This section contains sample reports for some of the multicorner-multimode reporting commands.

report_qor

The `report_qor` command reports by default the QoR details for all the scenarios in the design. The following example shows a report generated by the `report_qor` command:

```
****************************************
Report : qor
Design : DESIGN1
****************************************
  Scenario 's1'
  Timing Path Group 'reg2reg'
  -----------------------------------
  Levels of Logic:              33.00
  Critical Path Length:        694.62
  Critical Path Slack:        -144.52
  Critical Path Clk Period:    650.00
  Total Negative Slack:      -4533.01
  No. of Violating Paths:      136.00
  -----------------------------------
  Scenario 's2'
  Timing Path Group 'reg2reg'
  -----------------------------------
  Levels of Logic:              33.00
  Critical Path Length:        393.61
  Critical Path Slack:          61.18
  Critical Path Clk Period:    500.00
  Total Negative Slack:          0.00
  No. of Violating Paths:        0.00
  -----------------------------------
```

report_timing -scenario

This command reports timing results for the active scenarios in the design. You can specify a list of scenarios with the `-scenario` option. When the `-scenario` option is not specified only the current scenario is reported.

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : DESIGN1
Version: C-2009.06
Date   : Thu Apr 16 20:55:59 2009
****************************************

* Some/all delay information is back-annotated.

# A fanout number of 1000 was used for high fanout net computations.

Startpoint: TEST_BUF2En
              (input port clocked by clk)
Endpoint: TEST1/TEST2_SYN/latch_3
            (non-sequential rising-edge timing check clocked by clk)
Scenario: s1
Path Group: clk
Path Type: max
Point                                     Incr       Path      Lib:OC
-----------------------------------------------------------------------------
clock clk (rise edge)                     0.00       0.00
clock network delay (propagated)          0.00       0.00
input external delay                    450.00     450.00 f
TEST_BUF2En (in)                          0.00     450.00 f  stdcell_typ:WORST
TEST_BUF2En_BUF1/Z (inx4)                 9.75     459.75 r  stdcell_typ:WORST
U468/Z (inx10)                           10.21     469.96 f  stdcell_typ:WORST
TEST_BUF2En_BUF/Z (inx11)                 8.74     478.70 r  stdcell_typ:WORST
U293/Z (inx11)                            9.30     488.00 f  stdcell_typ:WORST
TEST1/TEST2_SYN/U74963/Z (nr2x4)         12.78     500.78 r  stdcell_typ:WORST
U31662/Z (inx4)                          10.58     511.37 f  stdcell_typ:WORST
TEST1/TEST2_SYN/U75093/Z (aoi21x6)       18.98     530.34 r  stdcell_typ:WORST
U42969/Z (nd2x6)                         14.16     544.51 f  stdcell_typ:WORST
TEST1/TEST2_SYN/U53046/Z (inx8)          13.35     557.86 r  stdcell_typ:WORST
U2765/Z (inx8)                           11.48     569.33 f  stdcell_typ:WORST
U32442/Z (inx6)                           7.61     576.94 r  stdcell_typ:WORST
U33615/Z (nd2x3)                         18.14     595.09 f  stdcell_typ:WORST
U32269/Z (nd2x6)                          8.74     603.82 r  stdcell_typ:WORST
TEST1/TEST2_SYN/clk_gate/EN (cklan2x1)    0.00     603.82 r  stdcell_typ:WORST
data arrival time                                  603.82
```

```
clock clk (rise edge)                 650.00     650.00
clock network delay (propagated)        0.00     650.00
TEST1/TEST2_SYN/clk_gate/CLK (cklan2x1)
                                        0.00     650.00 r
library setup time                    -56.25     593.75
data required time                               593.75
--------------------------------------------------------------
data required time                               593.75
data arrival time                               -603.82
--------------------------------------------------------------
slack (VIOLATED)                                 -10.07
```

report_constraint

This command reports constraints for all active scenarios. Each scenario is reported separately. When used with the `-scenario` option, it reports constraints for a specified list of scenarios.

```
****************************************
Report : constraint
Design : DESIGN1
Scenarios: 0, 1
Version: C-2009.06
Date   : Thu Apr 16 20:55:59 2009
****************************************
                                               Weighted
Group (max_delay/setup)      Cost     Weight     Cost    Scenario
-----------------------------------------------------------------
CLK                         10.07      1.00     10.07    s1
in2out                     372.89      1.00    372.89    s1
in2reg                     199.73      1.00    199.73    s1
reg2out                    467.99      1.00    467.99    s1
reg2reg                    171.16      1.00    171.16    s1
default                      0.00      1.00      0.00    s1
CLK                         90.60      1.00     90.60    s2
in2out                     474.97      1.00    474.97    s2
in2reg                     166.88      1.00    166.88    s2
reg2out                    326.46      1.00    326.46    s2
reg2reg                      0.00      1.00      0.00    s2
default                      0.00      1.00      0.00    s2
-----------------------------------------------------------------
max_delay/setup                               4404.52

...
                                            Multi-Scenario
Constraint                                      Cost
-----------------------------------------------------
multiport_net                                   0.00 (MET)
min_capacitance                                 0.00 (MET)
max_transition                                 45.28 (VIOLATED)
max_fanout                                    150.00 (VIOLATED)
max_capacitance                                 0.00 (MET)
max_delay/setup                              4404.52 (VIOLATED)
critical_range                               4404.52 (VIOLATED)
min_delay/hold                                  0.00 (MET)
max_area                                   714233.56 (VIOLATED)
```

report_tlu_plus_files

This command reports the TLUPlus files associations; it shows each minimum and maximum TLUPlus and layer map file per scenario:

```
dc_shell-topo> current_scenario s1
Current scenario is: s1

dc_shell-topo> report_tlu_plus_files
Max TLU+ file: /snps/testcase/s1max.tluplus
Min TLU+ file: /snps/testcase/s1min.tluplus
Tech2ITF mapping file: /snps/testcase/tluplus_map.txt
```

report_scenarios

The `report_scenarios` command reports the scenario setup information for multicorner-multimode designs. This command reports all the defined scenarios. The scenario-specific information includes the technology library used, the operating condition, and the TLUPlus files. The following example shows a report generated by the `report_scenarios` command:

```
***************************************
Report : scenarios
Design : DESIGN1
scenario(s) : SCN1
Version: C-2009.06
Date   : Fri Apr 17 20:55:59 2009
***************************************

All scenarios (Total=4): SCN1 SCN2 SCN3 SCN4
All Active scenarios (Total=1): SCN1
Current scenario     : SCN1

Scenario #0: SCN1 is active.
Scenario options:
Has timing derate: No
Library(s) Used:
 technology library name (File: library.db)

Operating condition(s) Used:
  Analysis Type    : bc_wc
  Max Operating Condition: library:WCCOM
  Max Process    : 1.00
  Max Voltage    : 1.08
  Max Temperature: 125.00
  Min Operating Condition: library:BCCOM
  Min Process    : 1.00
  Min Voltage    : 1.32
  Min Temperature: 0.00

Tlu Plus Files Used:
```

```
Max TLU+ file: tlu_plus_file.tf
Tech2ITF mapping file: tf2itf.map
```

report_power

The `report_power` command supports the `-scenario` option. Without the `-scenario` option, only the current scenario is reported. To report power information for all scenarios, use the `report_power -scenarios [all_scenarios]` command.

Note:
In the multicorner-multimode flow, the `report_power` command does not perform clock tree estimation. The command reports only the netlist power in this flow.

The following example shows the report generated by the `report_power -scenario` command.

```
****************************************
Report : power
Design : Design_1
Scenario(s): s1
Version: C-2009.06
Date   : Wed Apr 15 12:52:02 2009
****************************************

Library(s) Used: slow (File: slow.db)

Global Operating Voltage = 1.08
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = Unitless

Warning: Could not find correlated power. (PWR-725)

Power Breakdown
---------------

                          Cell       Driven Net  Tot Dynamic    Cell
                          Internal    Switching  Power (mW)     Leakage
Cell                            Ower (mW)     Power (mW)   (% Cell/Tot)  Power(nW)
------------------------------------------------------------------------
Netlist Power               4.8709     1.2889  6.160e+00 (79%) 1.351e+05
Estimated Clock Tree Power   N/A         N/A        (N/A)        N/A
------------------------------------------------------------------------
```

Supported SDC Commands

Table 10-17 lists the SDC commands supported in the multicorner-multimode flow.

Table 10-17 Supported SDC Commands

Commands	
all_clocks	set_fanout_load
create_clock	set_input_delay
create_generated_clock	set_input_transition
get_clocks	set_latency_adjustment_options
group_path	set_load
set_annotated_delay	set_max_capacitance
set_capacitance	set_max_delay
set_case_analysis	set_max_dynamic_power
set_clock_gating_check	set_max_leakage_power
set_clock_groups	set_max_time_borrow
set_clock_latency	set_max_transition
set_clock_transition	set_min_delay
set_clock_uncertainty	set_multicycle_path
set_data_check	set_output_delay
set_disable_timing	set_propagated_clock
set_drive	set_resistance
set_false_path	set_timing_derate

Multicorner-Multimode Script Example

Example 10-51 shows a basic sample script for the multicorner-multimode flow.

Example 10-51 Basic Script to Run a Multicorner-Multimode Flow

```
#......path settings......
set search_path ". $DESIGN_ROOT $lib_path/dbs \
           $lib_path/mwlibs/macros/LM"
set target_library "stdcell.setup.ftyp.db \
           stdcell.setup.typ.db stdcell.setup.typhv.db"
set link_library [concat * $target_library \
           setup.ftyp.130v.100c.db setup.typhv.130v.100c.db \
           setup.typ.130v.100c.db]
set_min_library stdcell.setup.typ.db -min_version stdcell.hold.typ.db

#......MW setup......
#......load design......

create_scenario s1
set_operating_conditions WORST -library stdcell.setup.typ.db:stdcell_typ
set_tlu_plus_files -max_tluplus design.tlup -tech2itf_map layermap.txt
read_sdc s1.sdc
set_max_leakage_power 0

create_scenario s2
set_operating_conditions BEST -library stdcell.setup.ftyp.db:stdcell_ftyp
set_tlu_plus_files -max_tluplus design.tlup -tech2itf_map layermap.txt
read_sdc s2.sdc

create_scenario s3
set_operating_conditions NOM -library stdcell.setup.ftyp.db:stdcell_ftyp
set_tlu_plus_files -max_tluplus design.tlup -tech2itf_map layermap.txt
read_sdc s3.sdc

set_active_scenarios {s1 s2}
report_scenarios
compile_ultra -scan -gate_clock
report_qor
report_constraints
report_timing -scenario [all_scenarios]
.
.
insert_dft
.
.
compile_ultra –incr -scan
```

Using ILMs in Multicorner-Multimode Designs

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 model contains cells whose timing is affected by or affects the external environment of a block. ILMs enhance capacity and reduce runtime for the optimization of the top-level design. For ILM details, see "Using Interface Logic Models" in Chapter 9.

ILMs are compatible with multicorner-multimode scenarios. You can apply multicorner-multimode constraints to an ILM and use the ILM in a top-level design.

The following requirements apply to using ILMs with multicorner-multimode scenarios:

- For each scenario in the top-level design, an identically named scenario must exist in each of the ILM blocks used in the top-level design. An ILM can have additional scenarios that are not used at the top-level design.
- If a top-level design does not have multicorner-multimode scenarios defined in it, the ILMs also cannot have multicorner-multimode scenarios defined in it.
- For each TLUPlus file that is used, the ILM stores the extraction data and the specified operating condition. In the top-level design, you cannot use additional TLUPlus files or define additional temperature corners for the existing TLUPlus files.

Methodology for Using ILMs With Scenarios at the Top Level

Follow these steps to use an ILM at the top level when it has scenario information:

1. Set the current design to the top-level design.
2. Remove all scenarios.

   ```
   remove_scenarios -all
   ```

3. Define scenarios for the top-level design.

   The scenarios defined in the top-level design must match the scenario definitions in the ILM blocks of the design. In the top-level design, all the scenarios must be defined before the next step.

4. Perform optimization.

   ```
   compile_ultra
   ```

At the beginning of compilation, the `compile_ultra` command performs the following sanity checks to ensure that there are no scenario mismatches between the top-level design and the ILMs. The compilation is terminated when any of the following mismatches are encountered:

- The number of scenarios in the top-level design match the number of scenarios in the ILM blocks.

  If the tool detects that the top-level design has more scenarios than ILM blocks, an ILM-70 error message is issued, and compilation is terminated.

  ```
  Error: Scenario S6 is not available in ILM Block1. (ILM-70)
  ```

- The scenario information in the top-level design is consistent with the scenario information in the ILM blocks.

  If scenarios are not defined in the top-level design and the ILM blocks have scenario definitions, an ILM-73 error message is issued and compilation is terminated.

  ```
  Error: Inconsistent use of of ILM BlockInit in the
  multicorner-multimode flow. ILM BlockInit has scenarios defined while
  top design Top does not have scenarios defined. (ILM-73)
  ```

You can also use the `check_scenarios` command to check consistency between scenarios. For more details, see the command man page.

11

Using a Milkyway Database

You can write a Milkyway database within Design Compiler to use with other Synopsys Galaxy platform tools, such as IC Compiler. You do this by way of the `write_milkyway` commands. You can use a single Milkyway library across the entire Galaxy flow.

Note:
Design Compiler does not support the `read_milkyway` command.

When you use a Milkyway database, you do not need to use an intermediate netlist file exchange format such as Verilog or VHDL to communicate with other Synopsys Galaxy platform tools. Before you can use a Milkyway database within Design Compiler, you must prepare a design library and a reference library and know about the concepts and tasks described in these sections:

- Licensing and Required Files
- Invoking the Milkyway Tool
- About the Milkyway Database
- Guidelines for Using the Milkyway Databases
- Preparing to Use the Milkyway Database
- Writing the Milkyway Database
- Maintaining the Milkyway Design Library
- Setting the Milkyway Design Library for Writing an Existing Milkyway Database

Licensing and Required Files

The Milkyway-Interface license is already provided to Design Compiler users. This license is needed for the write_milkyway commands.

Table 11-1 and Table 11-2 list the files required for using a Milkyway database.

Table 11-1 Source Files

Source file	Description
.lib	Source for logic libraries

Table 11-2 Compiled Databases

Compiled databases	Description
.db	Logic library containing standard cell timing, power, function, test, and so forth
Milkyway library	Physical technology data, FRAM

Invoking the Milkyway Tool

You invoke the Milkyway tool by using the Milkyway -galaxy command, which checks out the Milkyway-Interface license.

About the Milkyway Database

The Milkyway database stores design data in the Milkyway design library and physical library data in the Milkyway reference library.

- Milkyway design library

  The Milkyway directory structure used to store design data—that is, the uniquified, mapped netlist and constraints—is referred to as the Milkyway design library. You specify the Milkyway design library for the current session by setting the mw_design_library variable to the root directory path.

- Milkyway reference library

The Milkyway directory structure used to store physical library data is referred to as the Milkyway reference library. Reference libraries contain standard cells, macro cells, and pad cells. For information on creating reference libraries, see the Milkyway documentation.

You specify the Milkyway reference library for the current session by setting the `mw_reference_library` variable to the root directory path. The order in the list implies priority for reference conflict resolution. If more than one reference library has a cell with the same name, the first reference library has precedence.

Guidelines for Using the Milkyway Databases

When you use the `write_milkyway` command, observe these guidelines.

- Make sure all the cells present in the Milkyway reference library have corresponding cells in the timing library. The port direction of the cells in the Milkyway reference libraries are set from the port direction of cells in the timing library. If cells are present in the Milkyway reference library but are not in the timing library, the port direction of cells present in the Milkyway reference library is not set.
- Run the `uniquify` command before you run `write_milkyway`.
- You must make sure the units in the logic library and the Milkyway technology file are consistent.

  The SDC file does not contain unit information. If the units in the logic library and Milkyway technology file are inconsistent, the `write_milkyway` command cannot convert them automatically. For example, if the logic library uses femtofarad as the capacitance unit and the Milkyway technology file uses picofarad as the capacitance unit, the output of `write_sdc` shows different net load values.

  In the following example, the capacitance units in the logic library and the Milkyway technology file are not consistent. The following `set_load` information is shown for the net gpdhi_word_d_21_ before `write_milkyway` is run:

  ```
  set_load 1425.15 [get_nets {gpdhi_word_d_21_}]
  ```

  After `write_milkyway` is run, the SDC file shows

  ```
  set_load 8.36909 [get_nets {gpdhi_word_d_21_}]
  ```

- Design Compiler is case-sensitive. However, you can use the tool in case-insensitive mode by doing the following before you run `write_milkyway`:
  - Prepare uppercase versions of the libraries used in the link library.
  - Use the `change_names` command to make sure the netlist is uppercased.

Preparing to Use the Milkyway Database

You use the Milkyway design library to specify physical libraries and save designs in Milkyway format. The inputs required to create a Milkyway design library are the Milkyway reference library and the Milkyway technology file.

To create a Milkyway design library, follow these steps:

1. Define the power and ground nets. For example, set the following variables:

```
set mv_power_net VDD
set mw_ground_net VSS
set mw_logic1_net VDD
set mw_logic0_net VSS
set mw_power_port VDD
set mw_ground_port VSS
```

   If you do not set these variables, power and ground connections are not made during execution of `write_milkyway`. Instead power and ground nets can get translated to signal nets.

2. Use the `create_mw_lib` command to create the Milkyway design library. For example,

```
create_mw_lib -technology $mw_tech_file \
 -mw_reference_library $mw_reference_library $mw_design_library_name
```

3. Use the `open_mw_lib` command to open the Milkyway library that you created. For example, enter

```
open_mw_lib $mw_design_library_name
```

4. (Optional) Use the `set_tlu_plus` command to attach TLU+ files. For example,

```
set_tlu_plus_files-max_tluplus $max_tlu_file \
                      -min_tluplus $min_tlu_file\
                      -tech2itf_map $prs_map_file
```

5. In subsequent dc_shell sessions, you use the `open_mw_lib` command to open the Milkyway library. If you are using TLUPlus files for RC estimation, use the `set_tlu_plus_files` command to attach these files. For example,

```
open_mw_lib $mw_design_library_name
set_tlu_plus_files-max_tluplus $max_tlu_file \
                      -min_tluplus $min_tlu_file\
                      -tech2itf_map $prs_map_file
```

Writing the Milkyway Database

To save the design data in a Milkyway design library, use the write_milkyway command. The write_milkyway command writes netlist and physical data from memory to Milkyway design library format.

To use the write_milkyway command, enter

```
dc_shell> write_milkyway [options] file_name
```

Table 11-3 Using the write_milkyway Command Options

To do this	Use this
Specify the name of the design file to write (required). If you omit this switch, the tool displays an error message Error: Required argument '-output' was not found (CMD-007)	-output
Overwrite the current version of the design file. If you omit this option, a new, additional file version is written. Use this switch to save disk space.	-overwrite

You must use the -output option to specify the file name.

```
dc_shell> write_milkyway -output file_name -overwrite
```

For more information, see the man page.

Example

To write design information from memory to a Milkyway library named testmw and name the design file TOP, enter

```
dc_shell> set mw_design_library testmw
dc_shell> write_milkyway -output TOP
```

Important Points About the write_milkyway Command

When you use the write_milkyway command, keep the following points in mind:

- You must run create_mw_lib before you run write_milkyway.
- If a design file already exists (that is, you ran write_milkyway more than once on the design with the same output directory), write_milkyway creates a new, additional design file and increments the version number. You must make sure you open the correct

version in Milkyway; by default Milkyway opens the latest version. To avoid creating an additional version, use the `-overwrite` switch to overwrite the current version of the design file and save disk space.

- The command does not modify in-memory data.
- Attributes present in the design in memory that have equivalent attributes in Milkyway are translated (not all attributes present in the design database are translated).
- A hierarchical netlist translated using `write_milkyway` retains its hierarchy in the Milkyway database.

Results of Running the write_milkyway Command

The `write_milkyway` command does the following:

- Creates a design file based on the netlist in memory and saves the design data for the current design in the file. The path for the design file is *design_dir*/CEL/*file_name*:*version*, where *design_dir* is the location you specified in `mw_design_library`.
- Re-creates the hierarchy preservation information.

Limitations When Writing Milkyway Format

The following limitations apply when you write your design in Milkyway format:

- The design must be mapped.

  Because the Milkyway format describes physical information, it supports mapped designs only. You cannot use the Milkyway format to store design data for unmapped designs.
- The design must not contain multiple instances.

  You must uniquify your design before saving it in Milkyway format. Use the `check_design -multiple_designs` command to report information related to multiply-instantiated designs.
- The `write_milkyway` command saves the entire hierarchical design in a single Milkyway design file. You cannot generate separate design files for each subdesign.
- When you save a design in Milkyway format, the `write_milkyway` command does not save the interface logic model (ILM) instances in the Milkyway design library. You must explicitly save each ILM. For details, see Chapter 9, “Using Interface Logic Models”.

Script to Set Up and Write a Milkyway Database

Example 11-1 is a script to set up and write a Milkyway database.

Example 11-1 Script to Set Up and Write a Milkyway Database

```
set search_path "search_path ./libraries"
set link_library "* max_lib.db"
set target_library "max_lib.db"

create_mw_lib -technology $mw_tech_file -mw_reference_library \
                $mw_reference_library $mw_lib_name
open_mw_lib $mw_lib_name
read_file -format ddc design.ddc
current_design TopDesign
link
write_milkyway -output myTop
```

Maintaining the Milkyway Design Library

To maintain your Milkyway design library (for example, to delete unneeded versions of your design), you must use the Milkyway tool. This tool is a graphical user interface (GUI) that enables manipulation of the Milkyway libraries.

To invoke the Milkyway tool, enter

```
% Milkyway -galaxy
```

For information about using the Milkyway tool, see the Milkyway documentation.

Setting the Milkyway Design Library for Writing an Existing Milkyway Database

If the Milkyway design library already exists, run the `set_mw_design` command before you run the `write_milkyway` command to set the Milkyway design library directory.

When you use `set_mw_design`, keep the following points in mind:

- The Milkyway design library must already exist (created previously by `create_mw_lib`).
- Specify the `mw_reference_library` and `mw_design_library` variables. The `mw_reference_library` variable is used to set the Milkyway reference libraries for the design and to enhance the `search_path` variable.

12

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:

- Instantiating RTL PG Pins in a Non-UPF Mode
- Resolving Bus Versus Bit-Blasted Mismatches Between the RTL and Macros
- Fixing Errors Caused by New Unsupported Technology File Attributes
- Using Register Replication to Solve Timing QoR, Congestion, and Fanout Problems
- Comparing Design Compiler Topographical and IC Compiler Environments
- Assessing Design and Constraint Feasibility in Mapped Designs
- Checking for Design Consistency
- Analyzing Your Design During Optimization
- Analyzing Design Problems
- Analyzing Area
- Analyzing Timing

- Resolving Specific Problems

Instantiating RTL PG Pins in a Non-UPF Mode

Design Compiler can accept RTL designs containing a small number of power/ground pin connections on macros in non-UPF mode. The tool does not support a full PG netlist for a block. For example, the tool only supports designs that contain a small number of analog macros that have PG pins.

To instantiate PG pins in your RTL design, set the `dc_allow_rtl_pg` variable to `true`. The default is `false`. To preserve the PG connections in a Verilog output, execute the `write_file -pg -format verilog` command. To preserve the PG connections in a .ddc format output, execute the `write_file -format ddc` command. Note that when saving the design in .ddc format, you do not need to use the `-pg` option. When reading the .ddc file back into Design Compiler, you make sure that the `dc_allow_rtl_pg` variable is set to `true`, otherwise the tool issues a DDC-21 error:

```
Error: The feature used to generate this DDC file is not supported by

this tool or is not enabled in the current session. (DDC-21)

Information: To pass this netlist from DC to other tools, please use

write_file -pg -f verilog ...
```

To pass the design netlist to IC Compiler or Formality, you must use a Verilog output.

To use PG pins in your RTL design, observe the following guidelines:

- PG libraries are required
- FRAM must always have correct PG information
- The tool will not display the PG nets and pins; the `get_pins` command will not show PG pins
- The RTL design must represent all PG pins as wires, not as supply0, supply1, and so on
- The RTL design must instance PG pins by name, such as ref U1 (.pin(net), …);
- PG nets should reach the top level, but do not have to connect to top-level ports
- The tool will mark cells with PG pins with the `dont_touch` attribute
- If any UPF commands are executed, the derived PG network is converted to UPF and the UPF flow is followed

Example 12-1 shows Verilog RTL code that instantiates two PG pins: my_vdd and my_vss.

Example 12-1 Coding PG Pins in the RTL Design

```
module  my_design(a, b, c, my_vdd, my_vss);
input a, b, my_vdd, my_vss;
output c;
   my_macro  U1(.a(a), .b(b), .c(c), .VDD(my_vdd), .VSS(my_vss));
endmodule
```

Resolving Bus Versus Bit-Blasted Mismatches Between the RTL and Macros

Typically, the RTL pin names and the logical library pin names match. Signals are defined one way in both the RTL and the library. They are defined as busses or the bus is defined by its individual wires. Occasionally, mismatches occur during development when, for example, you port from one technology to another, or when the libraries and the RTL are in flux. When mismatches occur, you can set the `enable_bit_blasted_bus_linking` variable to `true` and read your design into Design Compiler, even though your RTL and libraries do not match with respect to bus verses bit-blasted pin names. When the `enable_bit_blasted_bus_linking` variable is set to `true`, the linker rules are relaxed such that you can read your design when mismatched pin names occur. The default value for the `enable_bit_blasted_bus_linking` variable is `false`.

Note that when the `enable_bit_blasted_bus_linking` variable is set to `true`, the tool will match names in accordance with the `bus_inference_style` and `bus_inference_descending_sort` variable settings.

Fixing Errors Caused by New Unsupported Technology File Attributes

Occasionally, IC Compiler adds support for new technology file attributes that are not yet supported in Design Compiler. In these cases, a TFCHK-009 error message is issued in Design Compiler but not in IC Compiler when using the same technology file.

If you get TFCHK-009 errors when reading in your technology file, check the spelling of the attributes and make sure that the attributes are spelled correctly. If you still have TFCHK-009 errors, you can use either of the following two methods to remove the TFCHK-009 errors. Before using either of the following methods, make sure that you can safely remove or ignore the new attributes without impacting the tool functionality. In most cases, new attributes are associated with new routing rules and should not impact the functionality of Design Compiler.

- Remove the new attributes from the specified section in the technology file.

- If the new attributes are safe to ignore, you can set the `ignore_tf_error` variable to `true` which enables the tool to ignore the unsupported attributes. This allows the tool to ignore all TFCHK-009 errors, but the errors will still be issued in the log file.

Using Register Replication to Solve Timing QoR, Congestion, and Fanout Problems

Design Compiler can replicate registers to address timing quality of results (QoR), congestion, and fanout issues. This feature is supported in both Design Compiler topographical mode and wire-load mode. To enable register replication, use the `set_register_replication` command.

For Design Compiler topographical, register replication is placement-aware, which can help reduce congestion in some cases. For wire-load mode, the load of the original replicated register is evenly distributed among the new replicated registers. For details, see the *Design Compiler Optimization Reference Manual*.

Comparing Design Compiler Topographical and IC Compiler Environments

Sometimes Design Compiler topographical and IC Compiler have different environment settings. These differences can lead to correlation problems. To help fix correlation issues between Design Compiler topographical and IC Compiler, use the `consistency_checker` command in your UNIX shell to compare the respective environments. However, before you can compare the two environments, you need to determine the environments of each tool. To do this, use the `write_environment` command as shown in Example 12-2 and Example 12-3. Use the `-output` option to specify the name of your output report. In Design Compiler, execute `write_environment` before `compile_ultra`; in IC Compiler, execute `write_environment` before `place_opt`. For command details, see the man pages.

Example 12-2 Using write_environment in Design Compiler

```
dc_shell-topo> write_environment -consistency -output DCT.env
```

Example 12-3 Using write_environment in IC Compiler

```
icc_shell> write_environment -consistency -output ICC.env
```

Once you have the environments for each tool, compare the environments using the `consistency_checker` command, as shown in Example 12-4.

Example 12-4 Comparing Design Compiler and IC Compiler Environments

```
unix_shell> consistency_checker -file1 DCT.env -file2 ICC.env \
                                -folder temp \
                                -html html_report | tee cc.log
```

The resulting HTML output report lists mismatched commands and variables. In Example 12-4, the HTML output report is written to the ./html_report directory. To access this report, open the ./html_report/index.html file in any Web browser. To reduce correlation problems, fix these mismatches before proceeding to optimization.

In addition to the HTML output report, the tool prints a summary report to the shell while the `consistency_checker` command is running. You can use the UNIX tee command, as shown in Example 12-4, to save this report to a file. In Example 12-4, the report is saved to the ./cc.log file.

Note:
The ./temp directory stores intermediate files the tool uses when executing the `consistency_checker` command. Once the command completes, you can remove the ./temp directory.

For more details, see SolvNet article 026366.

Assessing Design and Constraint Feasibility in Mapped Designs

To help debug missing timing constraints and assess design and constraint feasibility in mapped designs, use the `set_zero_interconnect_delay_mode` command, as shown in Example 12-5. In this example, you set the `set_zero_interconnect_delay_mode` command to `true`, run `report_qor`, and set the `set_zero_interconnect_delay_mode` command back to its default of `false` before proceeding with further optimization commands. When `set_zero_interconnect_delay_mode` is set to `true`, the tool analyzes your design with only cell delays and the capacitance of the pin-load on all the wires in the design to determine if your design meets timing goals. The tool does not consider wire capacitance due to timing paths.

Always set `set_zero_interconnect_delay_mode` back to its default of `false` before running your optimization step. The tool reports warning messages if you use optimization commands when `set_zero_interconnect_delay_mode` is `true`.

Example 12-5 set_zero_interconnect_delay_mode Sample Script

```
...
compile
...
set_zero_interconnect_delay_mode true
report_qor
set_zero_interconnect_delay_mode false
```

```
...
```

When set_zero_interconnect_delay_mode is set to true, the tool reports the following warning when you execute report_constraints or report_qor:

```
Warning: Timer is in zero interconnect delay mode. (TIM-177)
```

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.

Design Compiler runs the -check_design -summary command on all designs that are compiled; however, you can also use the command explicitly to verify design consistency. The command reports a list of warning and error messages.

- It reports an error if it finds a problem that Design Compiler cannot resolve. For example, recursive hierarchy (when a design references itself) is an error. You cannot compile a design that has check_design errors.
- 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.

  For a complete list of error and warning messages issued by the check_design command, see the manpage. Use options to the check_design command as follows:

Table 12-1 Using the check_design Command Options

To do this	Use this
To perform checks at only the current level of hierarchy (by default, the check_design command validates the entire design hierarchy)	-one_level
Disable warning messages	-no_warnings
Display a summary of warning messages instead of one message per warning	-summary

Table 12-1 Using the check_design Command Options (Continued)

To do this	Use this
Report information messages related to multiply instantiated designs. When you use this option, a list of multiply instantiated designs along with instance names and associated attributes (dont_touch, black_box, and ungroup) are displayed. By default, messages related to multiply instantiated designs are suppressed.	-multiple_designs
Suppress warning messages related to connection class violations before compile. Use this option when you are working on a GTECH design or netlist in which connection class violations are expected. Doing so improves runtime (especially if the GTECH design is large and has several connection class violations).	-no_connection_class

By default, during compile or execution of the check_design command, Design Compiler issues a warning message if a tri-state bus is driven by a non tri-state driver. You can have Design Compiler display an error message instead by setting the check_design_allow_non_tri_drivers_on_tri_bus variable to false. The default is true. When the variable is set to false, the compile command stops after reporting the error; however, the check_design command continues to run after the error is reported.

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 three tasks in which Design Compiler works to reduce the compile cost function:

- Delay optimization
- Design rule fixing
- 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 12-2 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 12-2 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.

Table 12-2 Compile Log Format Keywords (Continued)

Column	Column header	Keyword	Column description
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.
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 12-10. To checkpoint automatically between each phase of compile, set the compile_checkpoint_phases variable to true.

Analyzing Design Problems

Table 12-3 shows the design analysis commands provided by Design Compiler. For additional information about these commands, see the man pages.

Table 12-3 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.
Ports	report_port report_bus report_transitive_fanin report_transitive_fanout	Displays information about ports. Displays information about bused ports. Reports fanin logic. Reports fanout logic.
Nets	report_net report_bus report_transitive_fanin report_transitive_fanout	Reports net characteristics. Reports bused net characteristics. Reports fanin logic. Reports fanout logic.
Clocks	report_clock	Displays information about clocks.

Analyzing Area

Use the report_area command to display area information and statistics for the current design or instance. The command reports combinational, non-combinational, and total area. If you have set the current instance, the report is generated for the design of that instance; otherwise, the report is generated for the current design. Use the -hierarchy option to report area used by cells across the hierarchy.

Analyzing Timing

Use the report_timing command to generate timing reports for the current design or the current instance. By default, the command lists the full path of the longest maximum delay timing path for each path group. (Design Compiler groups paths based on the clock controlling the endpoint. All paths not associated with a clock are in the default path group. You can also create path groups by using the group_path command).

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 12-4 lists the timing analysis commands.

Table 12-4 Timing Analysis Commands

Command	Analysis task description
report_design	Shows operating conditions, wire load model and mode, timing ranges, internal input and output, and disabled timing arcs.
check_timing	Checks for unconstrained timing paths and clock-gating logic.
report_port	Shows unconstrained input and output ports and port loading.
report_timing_requirements	Shows all timing exceptions set on the design.
report_clock	Checks the clock definition and clock skew information.
report_path_group	Shows all timing path groups in the design.
report_timing	Checks the timing of the design.
report_constraint	Checks the design constraints.

Table 12-4 Timing Analysis Commands (Continued)

Command	**Analysis task description**
`report_delay_calculation`	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 12-6 shows a full path timing report with a large cell delay value.

Example 12-6 Full Path Timing Report

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : Adder8
Version: Y-2006.06
Date   : Mon May 1 10:56:49 2006
****************************************

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 12-7 shows the results of this command.

Example 12-7 Delay Calculation Report

```
****************************************
Report : delay_calculation
Design : Adder8
Version: Y-2006.06
Date   : Mon May 1 10:56:49 2006
****************************************

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 dctcl `get_cells` command to locate all unmapped components:

```
dc_shell> 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 `get_cells` command to locate all black box cells:

```
dc_shell> get_cells -hier -filter "is_black_box==true"
```

Finding Hierarchical Cells

All hierarchical cells have the `is_hierarchical` attribute. You can use the `get_designs` command to locate all hierarchical cells:

```
dc_shell> 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 12-8 shows the script that you can use to identify the scan pins in your technology library.

Example 12-8 Script to Identify Scan Pins

```
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 the following command:

```
dc_shell> 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 using the `set_disable_timing` command to disable the path trace.

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 “Design Rule Constraints” on page 7-3.

Use the `report_constraint` command to get details on constraint violations.

Figure 12-1 shows a design containing the IV1 cell.

Figure 12-1 Buffering Example

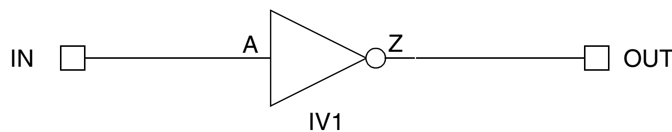


Table 12-5 gives the attributes defined in the technology library for the IV1 cell.

Table 12-5 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 12-9 shows the constraint report generated by the following command sequence:

```
set_drive 0 [get_ports IN]
set_load 0 [get_ports OUT]
report_constraint
```

Example 12-9 Constraint Report

```
****************************************
Report : constraint
Design : buffer_example
Version: Y-2006.06
Date   : Mon May 1 10:56:49 2006
****************************************

    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 12-10).

Example 12-10 Constraint Report (-verbose)

```
****************************************
Report : constraint
      -verbose
Design : buffer_example
Version: Y-2006.06
Date   : Mon May 1 10:56:49 2006
****************************************


    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 12-11).

Example 12-11 Constraint Report (-all_violators)

```
****************************************
Report : constraint
      -all_violators
Design : buffer_example
Version: Y-2006.06
Date   : Mon May 1 10:56:49 2006
****************************************

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 12-12 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 12-12 Constraint Report (After Port Attributes Are Modified)

```
****************************************
Report : constraint
           -all_violators
           -verbose
Design : buffer_example
Version: Y-2006.06
Date   : Mon May 1 10:56:49 2006
****************************************

    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 the following command:

```
dc_shell> 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 12-12:

```
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 get_pins command. For example, to determine whether a pin has a fanout_load attribute, enter

```
dc_shell> 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 get_pins command to check the capacitance values in the technology library:

```
dc_shell> 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.

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
- Read Script
- Compile Scripts

You can access the files described in these sections at $SYNOPSYS/doc/syn/guidelines, where $SYNOPSYS is the path to the installation directory.

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

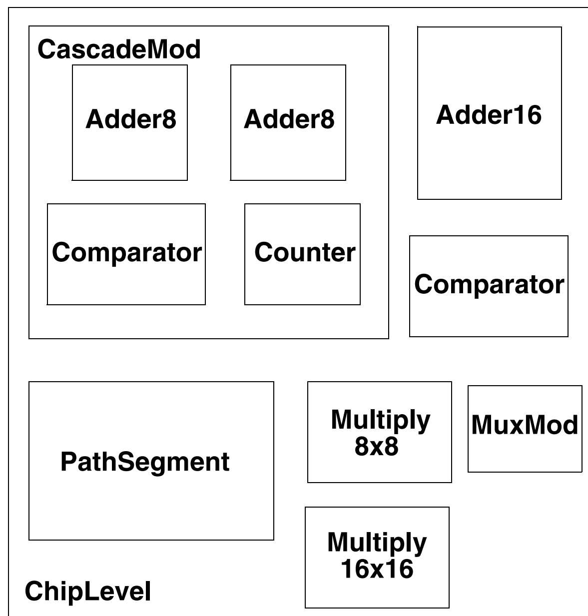
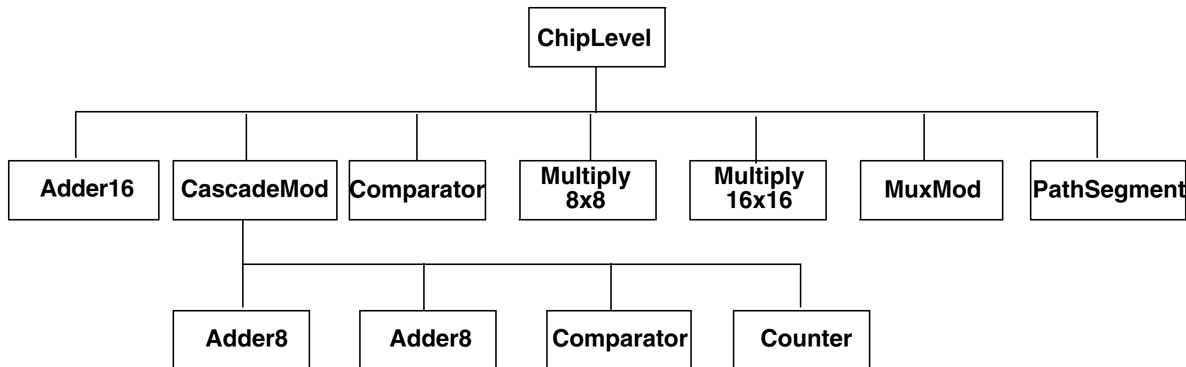


Figure A-2 shows the hierarchy for the design example.

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, bin;
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
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 the dctcl command language. For more information about the Tcl subset, see *Using Tcl With Synopsys Tools* and the *Design Compiler Command-Line Interface Guide*.

For details on the synthesis setup files, see "Setup Files" on page 2-5.

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 ddc_path "./ddc/"
set db_path "./db/"
set netlist_path "./netlist/"
```

Default Constraints File

The file shown in Example A-13 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

```
# 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 -lib_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
```

Read Script

Example A-15 provides the dctcl script used to read in the ChipLevel design.

The `read.tcl` script reads design information from the specified Verilog files into memory.

Example A-14 read.tcl

```
read_file -format verilog ChipLevel.v
read_file -format verilog Adder16.v
read_file -format verilog CascadeMod.v
read_file -format verilog Adder8.v
read_file -format verilog Counter.v
read_file -format verilog Comparator.v
read_file -format verilog Multiply8x8.v
read_file -format verilog Multiply16x16.v
read_file -format verilog MuxMod.v
read_file -format verilog PathSegment.v
```

Compile Scripts

Example A-15 through Example A-26 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 dctcl script files have the .tcl suffix. Scripts generated by the `write_script` command have the .wtcl suffix.

Example A-15 run.tcl

```
# Initial compile with estimated constraints
source "${script_path}initial_compile.tcl"

current_design ChipLevel
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}ChipLevel_init.db"
} else {
write -format ddc -hier -o "${ddc_path}ChipLevel_init.ddc"}

# 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
if {[shell_is_in_xg_mode]==0}{
write -hier -out "${db_path}ChipLevel_final.db"
} else {
write -format ddc -hier -out "${ddc_path}ChipLevel_final.ddc"}
```

Example A-16 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-17 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 -lib_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

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -format ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
```

Example A-18 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 -lib_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

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -format ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
```

Example A-19 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 -lib_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

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -format ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
```

Example A-20 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 -lib_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

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -format ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
```

Example A-21 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 -lib_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

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -format ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
report_timing_requirements -ignore \
   >> "${log_path}${rpt_file}"
```

Example A-22 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 -lib_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

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -format ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
```

Example A-23 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 -lib_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

if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}${design}.db"
} else {
write -format ddc -hier -o "${ddc_path}${design}.ddc"}

source "${script_path}report.tcl"
report_timing_requirements -ignore \
   >> "${log_path}${rpt_file}"
```

Example A-24 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-25 recompile.tcl

```
source "${script_path}read.tcl"

current_design ChipLevel
source "${script_path}defaults.con"

source "${script_path}adder16.wtcl"
compile
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}adder16_wtcl.db"
} else {
write -format ddc -hier -o "${ddc_path}adder16_wtcl.ddc"}
set rpt_file adder16_wtcl.rpt
source "${script_path}report.tcl"

source "${script_path}cascademod.wtcl"
dont_touch u12
compile
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}cascademod_wtcl.db"
} else {
write -format ddc -hier -o "${ddc_path}cascademod_wtcl.ddc"}
set rpt_file cascade_wtcl.rpt
source "${script_path}report.tcl"
source "${script_path}comp16.wtcl"
compile
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}comp16_wtcl.db"
} else {
write -format ddc -hier -o "${ddc_path}comp16_wtcl.ddc"}
set rpt_file comp16_wtcl.rpt
source "${script_path}report.tcl"

source "${script_path}mult8.wtcl"
compile
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}mult8_wtcl.db"
} else {
write -format ddc -hier -o "${ddc_path}mult8_wtcl.ddc"}
set rpt_file mult8_wtcl.rpt
source "${script_path}report.tcl"

source "${script_path}mult16.wtcl"
compile -ungroup_all
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}mult16_wtcl.db"
} else {
write -format ddc -hier -o "${ddc_path}mult16_wtcl.ddc"}
set rpt_file mult16_wtcl.rpt
source "${script_path}report.tcl"
report_timing_requirements -ignore \
           >> "${log_path}${rpt_file}"
```

```
source "${script_path}muxmod.wtcl"
compile
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}muxmod_wtcl.db"
} else {
write -format ddc -hier -o "${ddc_path}muxmod_wtcl.ddc"}
set rpt_file muxmod_wtcl.rpt
source "${script_path}report.tcl"


source "${script_path}pathseg.wtcl"
compile
if {[shell_is_in_xg_mode]==0}{
write -hier -o "${db_path}pathseg_wtcl.db"
} else {
write -format ddc -hier -o "${ddc_path}pathseg_wtcl.ddc"}
set rpt_file pathseg_wtcl.rpt
source "${script_path}report.tcl"
report_timing_requirements -ignore \
   >> "${log_path}${rpt_file}"
```

Example A-26 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}${rpt_file}"
report_qor >> "${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 the internal representation of the current design for consistency and issues error and warning messages as appropriate.

`check_timing`

Checks the timing attributes placed on the current design.

`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}
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
driving_cell_pin_fall	string
driving_cell_pin_rise	string
driving_cell_rise	string
fall_drive	float

Table C-2 Design Attributes (Continued)

Attribute name	Value
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
local_link_library	design_or_lib_file_name
max_capacitance	float
max_fanout	float
max_time_borrow	float
max_transition	float

Table C-2 Design Attributes (Continued)

Attribute name	Value
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

Table C-3 Library Attributes (Continued)

Attribute name	Value
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}
wired_or	{true, false}

Table C-6 Pin Attributes

Attribute name	Value
disable_timing	{true, false}

Table C-6 Pin Attributes (Continued)

Attribute name	Value
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.

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

```
hold = max clock delay - 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 specfied 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

```
setup = max data delay - 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.

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.

The uniquify process creates 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).

Index

A

accessing help 2-9
all_clocks command 5-17
all_connected command 5-31
all_outputs command 5-17
all_registers command 5-17
all_rp_groups command 10-61
all_rp_hierarchicals command 10-61
all_rp_inclusions command 10-61
all_rp_instantiations command 10-61
all_rp_references command 10-61
analyze command 2-17, 5-6, 5-10
analyzing design 12-10
  area 12-10
  timing 12-11
annotated load 7-26
architectural optimization 8-2
area
  maximum
    constraints for 7-15
  set maximum 7-18
area, voltage 6-18
async_set_reset compiler directive 3-12
asynchronous paths
  optimization constraints 7-16, 7-17
  set_max_delay command 7-16
  set_min_delay command 7-16, 7-17
attribute values
  saving 5-42
  setting 5-41
  viewing 5-42
attributes
  cell_degradation 7-11
  creating 5-43
  defined 5-40
  dont_touch 9-12
  getting descriptions 5-40
  is_interface_model 9-6, 9-9
  listing 5-27
  max_capacitance 7-9
  max_dynamic_power 7-3
  max_fanout 7-7
  max_leakage_power 7-3
  max_transition 7-5
  min_capacitance 7-10
  relative placement, writing to disk 10-63
  removing 5-27, 5-43
  search order 5-42
  set_auto_disable_drc_net 7-14
  viewing 12-23
attributes, list of
  auto_wire_load_selection 6-8
  clock C-1
  default_wire_load 6-8

default_wire_load_mode 6-9
design C-2
dont_touch 8-17, 12-16, 12-23, 12-24
is_black_box 12-15
is_hierarchical 12-15
is_unmapped 12-14
library C-4
library cell C-5
max_capacitance 12-24
max_fanout 12-24
max_transition 12-24
net C-5
pin C-5
reference C-6
auto_wire_load_selection attribute 6-8
automatic ungrouping 5-28
using compile 5-28

B

balance_buffer command 8-23
bottom-up compile 8-7
advantages 8-7
directory structure
figure 3-3
disadvantages 8-7
process 8-7
when to use 8-7
bound
order of precedence 10-40
using effectively 10-40
boundary conditions
subdesign 7-24
boundary optimization 8-26
bounding box
guidelines 10-40
breaking, feedback loop 12-16
buffers
extra 12-24
guidelines for working with 12-21
hanging 12-24
insertion process 12-17
interblock 12-16
missing 12-21
buses
creating 5-31
deleting 5-31

C

calculations, characterize command 7-26
capacitance
calculating 7-9
checking 12-22
control indirectly 7-8
limit directly 7-9
maximum 7-8, 7-9
control directly 7-8
frequency-based 7-9
minimum 7-10
capacitive load
setting 6-12
case sensitive
setting 5-14
case statement 3-18
latch inference 3-18
multiplexer inference 3-11
cell degradation 7-11
cell delays, finding source of 12-13
cell_degradation attribute 7-11
cells
black box, identifying 12-15
creating 5-31
deleting 5-31
grouping
from different subdesigns 5-30
from same subdesign 5-22
hierarchical
defined 5-4
identifying 12-15
leaf 5-4
library, specifying 4-9

listing 5-17
merging
hierarchy 5-30
reporting 5-17
unmapped, identifying 12-14
change_link command 5-15
change_names command 5-26, 5-37
characterize
subdesign port signal interfaces 7-29
combinational design 7-30
sequential design 7-31
subdesigns 7-22
characterize command 7-24
calculations 7-26
limitations 7-25
check_design command 12-6
check_design_allow_non_tri_drivers_on_tri_bus variable 12-7
checkpointing
defined 12-9
clock attributes C-1
clock skew
characterize command 7-25
clock trees
automatically disabling DRC fixing 7-14
clock-based maximum transition,specifying 7-5
clocks
create_clock command 7-16
listing 5-17
reporting 5-17
virtual 7-34
combinational logic
partitioning 3-5
command
analyzing design problems B-4
design constraints
setting B-3
design environment B-2
design rules B-2
report_reference 5-5
resolving design problems B-4
command language
dctcl 2-5
command log files 2-10
command script 2-10
commands
all_clocks 5-17
all_connected 5-31
all_outputs 5-17
all_registers 5-17
all_rp_groups 10-61
all_rp_hierarchicals 10-61
all_rp_inclusions 10-61
all_rp_instantiations 10-61
all_rp_references 10-61
analyze 2-17, 5-6, 5-10
analyzing design 12-10
balance_buffer 8-23
change_link 5-15
change_names 5-26, 5-37
characterize 7-24
calculations 7-26
check_design 12-6
compare_interface_timing 9-11, 9-14, 9-15, 10-78, 10-79
compile -auto_ungroup area 5-28
compile -auto_ungroup delay 5-28
compile_auto_ungroup delay 8-24
compile_ultra 2-19, 8-20
connect_net 5-31
connect_pin 5-31
copy_design 5-19
create_bounds 10-38
create_bus 5-31
create_cell 5-31
create_clock 7-16
create_design 5-19
create_ilm 9-9
create_multibit 3-13
create_net 5-29, 5-31
create_port 5-29, 5-31
current_design 5-12

current_instance 5-17
define_name_rules -map 5-37
disconnect_net 5-31
elaborate 2-17, 5-6, 5-10
exit 2-8
filter 12-14, 12-15
get_attribute 5-42, 12-22, 12-23
get_cells 12-14, 12-15
get_designs 12-15
get_ilm_objects 9-17, 9-19
get_ilms 9-19
get_license 2-13
get_magnet_cells 10-67
get_references 5-5
group 5-22, 8-23
group_path 8-21
gui_start 2-8
gui_stop 2-9
license_users 2-12
list 6-9
list_designs 5-11
list_instances 5-17
list_libs 4-8, 6-7
load_of 12-22
magnet_placement 10-66
max_dynamic_power 7-3
max_leakage_power 7-3
quit 2-8
read_db 5-11
read_ddc 5-11
read_file 2-17, 4-8, 5-6, 5-10, 5-12
read_floorplan 10-26
read_lib 4-8
read_verilog 2-17
read_vhdl 2-17
remove_attribute 7-7
remove_bus 5-31
remove_cell 5-31
remove_design 4-13, 5-34
remove_license 2-14
remove_multibit 3-13
remove_net 5-31
remove_port 5-31
remove_wire_load_model 6-9
rename_design 5-20
report_area 12-10
report_attribute 5-42
report_auto_ungroup 5-28
report_cell 12-23
report_clock 5-17
report_constraint 7-21, 10-126, 12-17
report_delay_calculation 12-13
report_design 6-4, 8-17
report_hierarchy 5-21
report_lib 6-3, 6-7, 6-12
report_net 5-17
report_port 5-17
report_qor 10-124
report_reference 5-17
report_timing 6-9, 12-11, 12-16
report_timing -scenario 10-125
report_tlu_plus_files 10-127
set_auto_disable_drc_nets 7-14
set_cell_degradation 7-11
set_cost_priority 7-20, 8-19
set_critical_range 8-22
set_disable_timing 12-15
set_dont_touch 8-14, 8-17
set_drive 6-10, 6-11
set_driving_cell 6-10, 6-12
set_equal 6-14
set_fanout_load 6-13, 7-8
set_input_delay 7-16
set_input_transition 6-10
set_load 6-12, 9-3
set_logic_dc 6-15
set_logic_one 6-16
set_logic_zero 6-16
set_max_area 7-18
set_max_capacitance 7-9
set_max_delay 7-16
set_max_fanout 7-7
set_max_leakage_power 10-105
set_max_transition 7-5

set_min_capacitance 7-10
set_min_delay 7-16
set_min_library 4-7, 10-113
set_mw_design 11-7
set_operating_conditions 10-110, 10-111
set_opposite 6-14
set_output_delay 7-16
set_resistance 12-16
set_tlu_plus_files 10-107
set_unconnected 6-17
set_ungroup 5-27, 8-27
set_wire_load 6-5, 6-9
translate 5-33
ungroup 5-25, 8-16
uniquify 8-13
write 5-35
write_floorplan 10-26
write_interface_timing 9-10, 9-11, 9-13, 9-14, 10-78, 10-79
write_lib 4-13
write_rp_groups 10-63
write_script 5-42
compare_interface_timing command 9-11, 9-14, 9-15, 10-78, 10-79
compile
default 8-19
defined 2-2
directory structure
bottom-up 3-3
top-down 3-3
high effort 8-24
incremental 8-25
subdesigns 7-22
compile -auto_ungroup area 5-28
compile -auto_ungroup delay 5-28
compile command
automatically uniquified designs 8-12
default behavior 8-19
disabling design rule cost function 8-19
disabling optimization cost function 8-19
compile cost function 8-18
compile log
customizing 12-8
compile script A-11
adder16 A-12
cascademod A-13
comparator A-14
multiply 8x8 A-15
multiply16x16 A-16
muxmod A-17
pathseg A-18
compile scripts
design example A-10, A-11
compile strategies 8-4
compile strategy
bottom-up 8-7
defined 2-18
mixed 8-11
top-down 8-5
compile_assume_fully_decoded_three_state_busses variable 5-34
compile_ultra command 2-19, 8-20
compiler directives
async_set_reset 3-12
enum 3-16
full_case 3-18
implementation 3-14
infer_multibit 3-13
infer_mux 3-11
label 3-14
map_to_module 3-14, 3-23
ops 3-14
return_port_name 3-23
state_vector 3-16
sync_set_reset 3-12
compiler_log_format variable 12-8
connect_net command 5-31
connect_pin command 5-31
constant nets
automatically disabling DRC fixing 7-14
constants
global

defining 3-19
constraints
boundary conditions 7-24
cost vector 7-19
design rule 7-3
cell degradation 7-11
maximum capacitance 7-8, 7-9
maximum fanout 7-6, 7-7
maximum transition time 7-4
minimum capacitance 7-10
maximum
area 7-15
maximum capacitance 7-8
maximum fanout 7-6
minimum capacitance 7-10
minimum delay 7-17
optimization 7-15
maximum area 7-18
maximum delay 7-16
minimum delay 7-17
timing 7-15
placement, setting 10-38
priorities 7-19
set_cost_priority command 7-20
simplifying 3-6
timing
asynchronous 7-15
synchronous 7-15
transition time 7-4
constraints file
design example A-10
copy_design command 5-19
cost calculation
maximum area 7-18
maximum delay 7-16
minimum delay 7-17
cost function 7-2, 8-18
constraints
report_constraint command 12-18
design rule 8-18
design rule constraints 7-2
design rules 7-4
equation 7-4
maximum delay equation 7-16
optimization 7-15, 8-18
optimization constraints 7-2
cost vector, constraints 7-19
create_bounds command 10-38
create_bus command 5-31
create_cell command 5-31
create_design command 5-19
create_ilm command 9-9
create_multibit command 3-13
create_net command 5-29, 5-31
create_port command 5-29, 5-31
critical-path resynthesis 8-24
current design
defined 5-3
displaying 5-12
current instance 5-4
changing 5-17
default 5-17
defined 5-17
displaying 5-18
resetting 5-18
current_design
command 5-12
variable 5-12
current_instance
command 5-17
variable 5-18

D

dangling logic, preserving 12-16
data management 3-2
data organization 3-2
.db format
reading 5-11
DC Expert
defined 1-4
DC Ultra

defined 1-4
dc_shell
exiting 2-8
session example 2-20
dctcl command language 2-5
.ddc format
reading 5-11
saving 5-36
default compile 8-19
default_wire_load attribute 6-8
default_wire_load_mode attribute 6-9
define_name_rules -map command 5-37
definitions
attribute 5-40
checkpointing 12-9
compiler 2-2
current design 5-3
current instance 5-4, 5-17
design 5-2
flat design 5-2
hierarchical cell 5-4
hierarchical design 5-2
leaf cell 5-4
nets 5-4
networks 5-4
optimization 2-2
parent design 5-2
pin 5-4
ports 5-4
subdesign 5-2
synthesis 2-1
delay
maximum 7-16
cost calculation 7-16
delay calculation, reporting 12-13
design
characterize 7-22
logical connections of ports 7-33
data management 3-2
in memory 5-1
organization 3-2
design attributes C-2
Design Compiler
description 1-1
design flow 1-2
exiting 2-8
family of products 1-3
help 2-9
interfaces 2-5
session example 2-20
starting 2-7
Design Compiler family
DC Expert 1-4
DC Ultra 1-4
Design Vision 1-5
DesignWare 1-5
DFT Compiler 1-5
HDL Compiler 1-5
Power Compiler 1-5
Design Compiler Topographical Technology 10-1
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-10, A-11
compile strategies for A-3
constraints file A-10
hierarchy A-3
setup file A-9
design exploration 8-19
basic flow 2-15
invoking 8-19
design files
reading 2-17, 5-6, 5-10
writing 5-35
design flow 1-2
high-level

figure 2-3
synthesis
design exploration 2-15
design implementation 2-15
design function
target libraries 4-3
design hierarchy
changing 5-21
displaying 5-21
preserved timing constraints 5-30
removing levels 5-25
See also, hierarchy
design implementation 8-20
basic flow 2-15
techniques for 8-20
design objects
accessing 5-17
adding 5-19
defined 5-2
listing
clocks 5-17
instances 5-17
nets 5-17
ports 5-17
references 5-17
registers 5-17
specifying
absolute path 5-18
relative path 5-17
design problems
commands
analyzing B-4
resolving B-4
design reuse
partitioning 3-4
design rule
constraints 7-3
precedence 7-12
max_capacitance attribute 7-9
max_transition attribute 7-5
maximum fanout
calculation 7-6
maximum transition time 7-4
min_capacitance attribute 7-10
priorities 7-12
design rule constraints
defined 4-3
design rule cost function 8-18
design rules
commands B-2
design rules cost function equation 7-4
Design Vision
defined 1-5
designs
analyzing 12-10
area 12-10
timing 12-11
checking consistency 12-6
copying 5-19
creating 5-19
current 5-3
defined 5-2
editing 5-31
buses 5-31
cells 5-31
nets 5-29, 5-31
ports 5-29, 5-31
flat 5-2
hierarchical 5-2
linking 4-5, 5-13
listing
details 5-12
names 5-11
listing current 5-12
parent 5-2
preserving implementation 8-17
reading 2-17, 5-6, 5-10
.db format 5-11
HDL (analyze command) 5-6
HDL (elaborate command) 5-7
netlists 2-17
RTL 2-17
reference, changing 5-15
removing from memory 5-34

renaming 5-20
reporting attributes 8-17
saving 5-35
default behavior 5-35
multiple 5-37
supported formats 5-35
translating 5-33
updating links for renamed designs 5-20
DesignWare
defined 1-5
DesignWare library
defined 1-5, 4-4
file extension 4-5
specifying 4-5, 4-7
device degradation 7-11
DFT Compiler
defined 1-5
directory structure
bottom-up compile
figure 3-3
top-down compile
figure 3-3
disabling
false violation messages 12-15
timing paths
scan chains 12-15
disabling DRC fixing automatically on clock trees and constant nets 7-14
disconnect_net command 5-31
domain, power 6-18
dont_touch attribute 9-12, 12-23, 12-24
and dangling logic 12-16, 12-24
and timing analysis 8-17
reporting, designs 8-17
setting 8-17
drive characteristics
removing 6-10
setting
command to 6-10
example of 6-11
drive resistance, setting 6-11
drive strength
defining 6-10

E

elaborate command 2-17, 5-6, 5-10
enable level shifter 6-18
environment variables
SNPS_MAX_QUEUETTIME 2-14
SNPS_MAX_WAITTIME 2-14
SNPSLMD_QUEUE 2-13
equation
design rule cost 7-4
examples of
ungrouping hierarchy 5-27
exit command 2-8
exiting Design Compiler 2-8
expressions
guidelines
HDL 3-22
extent 6-18

F

false violation messages, disabling 12-15
fanout
calculation 7-6
constraints 7-6
maximum 7-6
specifying values of 6-13
fanout load
determine 7-8
set 7-8
feedback loop
breaking 12-16
identifying 12-16
file name extensions
conventions 3-2
filename log files 2-10
filename_log_file variable 2-10
files

command log file 2-10
filename log file 2-10
script 2-10
flat design 5-2
flip-flop
defined 3-12
inferring 3-12
floorplan exploration
creating and editing floorplans 10-98
enabling 10-95
exiting 10-100
flow 10-95
limitations 10-101
overview 10-94
resynthesizing the design 10-101
saving floorplan for synthesis 10-100
saving into a Tcl script or DEF file 10-99
saving or discarding floorplan updates 10-99
setting floorplanning options 10-95
using 10-94
using the GUI 10-97
floorplans, creating and editing 10-94
frequency-based maximum capacitance 7-9
frequency-based maximum capacitance,specifying 7-9
full_case directive 3-18
functions
guidelines
HDL 3-23

G

gate-level optimization 8-3
get_attribute command 5-42, 12-22, 12-23
get_cells command 12-14, 12-15
get_designs command 12-15
get_ilm_objects command 9-17, 9-19
get_ilms command 9-19
get_license command 2-13
get_magnet_cells command 10-67
glue logic 3-5
group command 5-22, 8-23
group_path command
-critical_range option 8-22
features of 8-21
grouping
adding hierarchy levels 5-22
groups
relative placement, defined 10-49
GUI
opening 2-8
gui_start command 2-8
gui_stop command 2-9

H

HDL Compiler
defined 1-5
HDL design, reading
analyze command 5-6
elaborate command 5-7
help
accessing 2-9
hierarchical block
outside load 7-27
hierarchical boundaries
wire load model 6-5
hierarchical cells
defined 5-4
identifying 12-15
hierarchical compile
See, top-down compile
hierarchical designs
defined 5-2
reporting area across hierarchy 12-10
reporting references across hierarchy 5-5
hierarchical pin timing constraints, preserving 5-29
hierarchical pins, preserving timing constraints 5-29
hierarchy

adding levels 5-22
changing 5-21
changing interactively 5-22
displaying 5-21
merging cells 5-30
removing levels 5-25, 5-27, 8-27
all 5-25
ungrouping automatically 5-28
high-effort compile 8-24
hlo_disable_datapath_optimization variable 8-28
hold checks
timing arcs and 6-12

I

identifiers
guidelines
HDL 3-21
identifying
black box cells 12-15
feedback loops 12-16
hierarchical cells 12-15
unmapped cells 12-14
if statement 3-17
ILM (interface logic model) 9-1
ILMs
compare_interface_timing 9-11, 9-14, 9-15, 10-78, 10-79
write_interface_timing 9-10, 9-11, 9-13, 9-14, 10-78, 10-79
incremental compile 8-25
infer_multibit compiler directive 3-13
infer_mux compiler directive 3-11
inferring registers 3-12
input delay
characterize command 7-26
input port
always one or zero 6-15
set_equal command 6-14
set_logic_dc command 6-15
set_opposite command 6-14
instances
current 5-4
listing 5-17
reporting 5-17
interblock buffers 12-16
interface
graphical user interface 2-5
interface logic model
benefits of using 9-3
command summary, list of 9-19
defined 9-1
dont_touch attribute 9-12
fanins and fanouts of chip-level networks, controlling 9-6
flow for creating 9-9
guidelines for creating 9-4
information about, reporting 9-16
instantiating and using 9-12
logic included in, controlling 9-5
number of latch levels, controlling 9-7
overview 9-2
preserving as subdesign 8-17
set_load command 9-3
side-load cells, controlling 9-8
top-down compile 8-5, 8-6
interfaces
dc_shell 2-5
is_black_box attribute 12-15
is_hierarchical attribute 12-15
is_interface_model attribute 9-6, 9-9
is_unmapped attribute 12-14
isolation cell 6-18

K

k-factors
unsupported in multicorner-multimode 10-113

L

latches
  defined 3-12
  inferring 3-12
leaf cell 5-4
level shifter 6-18
libraries
  DesignWare 1-5, 4-4
  link 4-3
  list of 6-7
  list values of 6-7, 6-12
  listing
    names 4-8
  main 4-7
  power consumption 4-4
  reading 4-8
  removing from memory 4-13
  reporting contents 4-9
  saving 4-13
  specifying 2-17, 4-5
    objects 4-9
  symbol 4-4
  synthetic 3-13
  target 4-3
  technology 4-2
  timing values 4-4
library attributes C-4
library cell
  specifying 5-33
library cell attributes C-5
library objects
  defined 4-9
  specifying 4-9
library registers
  specifying 5-33
license queuing
  enabling 2-13
  environment variables 2-13
    SNPS_MAX_QUEUETIME 2-14
    SNPS_MAX_WAITTIME 2-14
    SNPSLMD_QUEUE 2-13
license_users command 2-12
licenses
  checking out 2-13
  enabling queuing 2-13
  listing 2-12
  releasing 2-14
  using 2-12
  working with 2-12
limitations
  interface logic models, saving 11-6
  Milkyway design library
    writing 11-6
link library
  file extension 4-5
  libraries
    cell references 4-3
  specifying 4-5
link_force_case variable 5-14
link_library variable 4-5, 5-14
list command 6-9
list_designs command 5-11
list_instances command 5-17
list_libs command 4-8, 6-7
load
  annotated 7-26
  characterize command 7-26
  outside 7-27
load_of command 12-22
log files
  command log file 2-10
  filename log file 2-10
logic-level optimization 8-3

M

magnet placement 10-66
magnet_placement command 10-66
main library 4-7
man pages
  accessing 2-9
max_capacitance attribute 7-9, 12-24

max_dynamic_power 7-3
max_fanout attribute 7-7, 12-24
max_leakage_power 7-3
max_transition attribute 7-5, 12-24
maximum area
  cost calculation 7-18
  equation 7-18
maximum capacitance
  frequency-based 7-9
maximum delay 7-16
  cost calculation 7-16
maximum fanout 7-6
maximum fanout calculation 7-6
maximum performance optimization 8-21
maximum timing information 4-7
maximum transition time 7-4
  clock-based 7-5
  set_max_transition command 7-5
messages
  disabling 12-15
Milkyway database
  guidelines for using 11-3
  Milkyway design library, creating 11-4
  preparation for reading 11-7
Milkyway design library
  defined 11-2, 11-3
  existing Milkyway database, preparation for reading 11-7
  maintaining 11-7
  specifying 11-2, 11-3
Milkyway format
  limitations
    writing 11-6
  limitations when writing 11-6
min_capacitance attribute 7-10
minimum area optimization 8-25
minimum capacitance 7-10
minimum delay 7-17
  cost calculation 7-17
  equation 7-17
minimum timing information 4-7
mixed compile strategy 8-11
model, interface logic, defined 9-1
modules
  guidelines
    HDL 3-24
multicore 2-11, 2-12
multicorner-multimode
  k-factors, unsupported 10-113
  report_constraint command 10-126
  report_qor command 10-124
  report_timing -scenario command 10-125
  report_tlu_plus_files command 10-127
  scenario management, commands 10-114
  scenario reduction 10-116
  script example 10-129
  set_min_library command 10-113
  set_tlu_plus_files command 10-107
  setup considerations 10-107
multimode
  scenario definition 10-103
  set_max_dynamic_power command 10-117
  set_max_leakage_power command 10-105
multiple instances of a design
  resolving 8-11
multiple instances, resolving
  compile command automatic uniquify 8-13
  compile-once-don’t-touch method 8-14
  ungroup method 8-16
  uniquify method 8-13
multiplexers
  inferring 3-11
    HDL Compiler 3-11
mw_logic0_net variable 10-10, 11-4
mw_logic1_net variable 10-10, 11-4

N

name
  changing net or port 5-37
naming conventions

- file name extensions 3-2
- library objects 4-9
- signal name suffixes 3-22

naming translation 5-37
net attributes C-5
net names
- changing name rules 5-37

net, supply 6-18
netlist
- editing 5-31
- reading 2-17

netlist reader 2-17
nets 5-4
- connecting 5-31
- creating 5-29, 5-31
- disconnecting 5-31
- heavily loaded, fixing 8-23
- reporting 5-17

networks 5-4

O

opening the GUI 2-8
operating conditions
- defining 6-3
- list of
  - current design 6-4
  - technology library 6-3

optimization
- across hierarchical boundaries 8-26
- architectural 8-2
- boundary 8-26
- constraints 7-15
  - asynchronous paths 7-16
  - Power Compiler 7-3
- cost function 8-18
- data paths 8-27
- defined 2-2
- gate level 8-3
- gate-level 8-3
- high-speed designs 8-20
- how it works 12-8
- incremental 8-25
- invoking 8-19
- logic-level 8-3
- maximum performance 8-21
- minimum area 8-25
- port settings 6-13
- priorities 7-15
- trials phase 12-8

optimization cost function 7-15
optimization processes 8-2
output formats
- supported 5-35

output port
- fanout load 7-8
- unconnected 6-17

outside load 7-27
overview of interface logic models 9-2

P

partitioning
- by compile technique 3-7
- combinational logic 3-5
- design reuse considerations 3-4
- glue logic 3-5
- merge resources 3-8
- modules by design goals 3-7
- modules with different goals 3-7
- random logic 3-7
- sharable resources 3-8
- structural logic 3-7
- user-defined resources 3-8

path groups
- creating 12-11

paths
- asynchronous
  - constraints for 7-15
- synchronous
  - constraints for 7-15
- using absolute 5-18

using relative 5-17
paths delay
override default
set_false_path command 7-17
set_multicycle_path command 7-17
pin attributes C-5
pins 5-4
connecting 5-31
library cell, specifying 4-9
relationship to ports 5-4
placement
magnet 10-66
placement bounding box
guidelines 10-40
placement constraint, setting 10-38
point-to-point exception 7-34
port
always one or zero 6-15
driving violation at 7-10
logical connections
characterize 7-33
names, changing 5-37
output
fanout load 7-8
unconnected 6-17
set_equal command 6-14
set_logic_dc command 6-15
set_opposite command 6-14
port, supply 6-18
ports 5-4
capacitive load on
setting 6-12
creating 5-29, 5-31
deleting 5-31
listing
output ports 5-17
relationship to pins 5-4
reporting 5-17
setting drive characteristics of 6-10, 6-11
wire delays, preventing 12-16
Power Compiler
defined 1-5
optimization constraints 7-3
power domain 6-18
extent 6-18
scope 6-18
power domains
definition 10-71
power state table 6-18
power switch 6-18
preserved timing constraints in design hierarchies 5-30
preserving subdesigns 8-17
priorities, constraints 7-19

Q

queuing licenses 2-13
quit command 2-8
quitting Design Compiler 2-8

R

read_db command 5-11
read_ddc command 5-11
read_file command 2-17, 4-8, 5-6, 5-10, 5-12
read_floorplan 10-26
read_lib command 4-8
read_verilog command 2-17
read_vhdl command 2-17
reference attributes C-6
references
changing design 5-15
reporting 5-5, 5-17
resolving 4-5, 5-13
using 5-5
register inference
D flip-flop 3-12
D latch 3-12
defined 3-12
edge expressions 3-12

register types
  mixing 3-12
registers
  inferring
    HDL Compiler 3-12
  listing 5-17
relative placement
  aligning by pins 10-55
  group
    creating 10-49
    defined 10-49
  hierarchical group
    included, defined 10-59
    instantiated, defined 10-59
  methodology 10-46
  positions for data 10-50
  sample script 10-65
  writing information 10-63
remove_attribute command 7-7
remove_bus command 5-31
remove_cell command 5-31
remove_design command 4-13, 5-34
remove_license command 2-14
remove_multibit command 3-13
remove_net command 5-31
remove_port command 5-31
remove_wire_load_model command 6-9
removing levels of hierarchy 5-25
rename_design command 5-20
report
  constraint 7-21
report_area command 12-10
report_attribute command 5-42
report_auto_ungroup 5-28
report_cell command 12-23
report_clock command 5-17
report_constraint command 7-21, 10-126, 12-17, 12-19
  -all_violators option
    report violations 12-19
  -verbose option 12-18
report_delay_calculation command 12-13
report_design command 6-4, 8-17
report_hierarchy command 5-21
report_lib command 4-9, 6-3, 6-7, 6-12
report_net command 5-17
report_port command 5-17
report_qor command 10-124
report_reference command 5-5, 5-17
report_timing command 12-11
  feedback loops 12-16
  wire load information 6-9
report_timing -scenario command 10-125
report_tlu_plus_files command 10-127
reports
  analyzing design 12-10
  check_design command 12-6
  delay calculation 12-13
  library contents 4-9
  operating condition 6-3
  operating conditions 6-4
  report_hierarchy command 5-21
  script file A-21
  timing path 12-13
  wire load model
    example 6-7
resistance
  output driver
    defining 6-10
  See also, drive characteristics
resolving multiple instances of a design 8-11
resources
  shareable 3-8
  user-defined
    partitioning 3-8
retention register 6-19
RTL, reading 2-17

S

scenario definition

multimode 10-103
scenario reduction, multicorner-multimode 10-116
scope 6-18
script files 2-10
compile A-11
executing 2-10
generating 5-42
report A-21
search path
for libraries 4-8
search_path variable 4-8
semiconductor vendor, selecting 4-2
sequential device, initialize or control state 3-12
set_auto_disable_drc_net attribute 7-14
set_auto_disable_drc_nets command 7-14
set_cell_degradation command 7-11
set_cost_priority command 7-20, 8-19
set_critical_range command 8-22
set_disable_timing command 12-15
set_dont_touch command 8-14, 8-17
set_drive command 6-10, 6-11
set_driving_cell command 6-10, 6-12
set_equal command 6-14
set_false_path command 7-17
set_fanout_load command 6-13, 7-8
set_fix_hold command
minimum delay cost 7-17
set_input_transition command 6-10
set_load command 6-12, 9-3
set_logic_dc command 6-15
set_logic_one command 6-16
set_logic_zero command 6-16
set_max_area command 7-18
set_max_capacitance command 7-9
undo 7-9
set_max_fanout command 7-7
set_max_transition command
undo 7-5
set_min_capacitance command 7-10
undo 7-11
set_min_delay command 7-17
set_min_library command 4-7, 10-113
set_mw_design command 11-7
set_operating_condtions command 10-110, 10-111
set_opposite command 6-14
set_resistance command 12-16
set_tlu_plus_files command 10-107
set_unconnected command 6-17
set_ungroup command 5-27, 8-27
set_wire_load command 6-5, 6-9
setup checks
timing arcs and 6-12
setup files
design example A-9
.synopsys_dc.setup file 2-5
signals, edge detection 3-12
specifying
libraries
DesignWare 4-7
link 4-5
symbol 4-5
target 4-5
library objects 4-9
wire load mode 6-9
wire load model 6-9
state machine design 3-15
statements
'define 3-19
case 3-18
constant 3-19
if 3-17
subdesign
port signal interfaces
characterize 7-29, 7-30, 7-31
wire load model
characterize 7-30
subdesigns 5-2

preserving 8-17
supply net 6-18
supply port 6-18
switch, power 6-18
symbol library
defined 4-4
file extension 4-5
search path for 4-8
specifying 4-5
symbol_library variable 4-5
sync_set_reset directive 3-12
synchronous paths
create_clock command 7-16
set_input_delay command 7-16
set_output_delay command 7-16
.synopsys_dc.setup file 2-5
sample 2-6
synthesis
defined 2-1
synthesis design flow
figure 2-16
synthetic libraries 3-13
synthetic_library variable 4-5

T

table
bounds commands, summary 10-41, 10-48
target library
definition 4-3
file extension 4-5
specifying 4-5
target_library variable 4-5
technology library
creating 4-3
definition 4-2
required format 4-3
search path for 4-8
timing
boundary conditions 7-24
point-to-point exception 7-34
timing analyzer
timing constraints 7-16
timing arcs
hold checks and 6-12
setup checks and 6-12
timing budgets
characterize command 7-25
timing constraints
analyzer 7-16
optimization 7-15
timing information
maximum 4-7
minimum 4-7
timing path, report 12-13
timing values
link libraries 4-4
timing violations
correcting 12-23
scan chain 12-15
top-down compile 8-5
advantages 8-6
directory structure
figure 3-3
Topographical technology 10-1
transition time
constraints 7-4
maximum 7-4
clock-based 7-5
translate command 5-33
translating designs
procedure for 5-33
restrictions 5-34

U

ungroup command 5-25, 8-16
ungroup design
compile option 5-27, 8-27
ungroup hierarchy
examples 5-27
ungroup_preserve_constraints variable 5-29

ungrouping
automatically during compile 5-28, 8-24
removing hierarchy levels 5-25, 5-27, 8-27
uniquify command 8-13
uniquify method 8-12

V

variables
check_design_allow_non_tri_drivers_on_tri_bus 12-7
compile_assume_fully_decoded_three_state_busses 5-34
compile_log_format 12-8
current_design 5-12
current_instance 5-18
filename_log_file 2-10
hlo_disable_datapath_optimization 8-28
link_force_case 5-14
link_library 4-5, 5-14
mw_logic0_net 10-10, 11-4
mw_logic1_net 10-10, 11-4
search_path 4-8
SNPS_MAX_QUEUETIME 2-14
SNPS_MAX_WAITTIME 2-14
SNPSLMD_QUEUE 2-13
symbol_library 4-5
synthetic_library 4-5
target_library 4-5
ungroup_preserve_constraints 5-29
Verilog
expressions 3-22
functions 3-23
identifiers 3-21
modules 3-24
VHDL
expressions 3-22
functions 3-23
identifiers 3-21
modules 3-24
violations at driving ports 7-10
voltage area 6-18

W

wire delays, on ports 12-16
wire load
defining 6-4
wire load mode
default 6-9
reporting 6-9
specifying 6-9
wire load model
automatic selection
described 6-8
disabling 6-8, 6-9
choosing 6-9
default 6-8
enclosed 7-30
hierarchical boundaries 6-5
list of
technology libraries 6-7
removing 6-9
report example 6-7
reporting 6-9
segmented 7-30
specifying 6-9
subdesign
characterize 7-30
top 7-30
wire_load_selection library function 6-8
write command 5-35
write_floorplan command 10-26
write_interface_timing command 9-10, 9-11, 9-13, 9-14, 10-78, 10-79
write_lib command 4-13
write_rp_groups command 10-63
write_script command 5-42
characterize command 7-26