

Power Compiler™

User Guide

Version C-2009.06-SP2, September 2009

SYNOPSYS®

Copyright Notice and Proprietary Information

Copyright © 2009 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, HDL Analyst, HSIM, HSPICE, Identify, iN-Phase, 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, Eclipse, Encore, EPIC, Galaxy, Galaxy Custom Designer, HANEX, HAPS, 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	xvii
About This User Guide	xviii
Customer Support.	xx
1. Introduction to Power Compiler	
Power Compiler Methodology	1-2
Power Library Models	1-3
Power Analysis Technology	1-3
Power Optimization Technology	1-5
Working With Power Compiler	1-6
Library Requirements	1-6
Command-Line Interface	1-6
License Requirements	1-7
Reading and Writing Designs	1-7
Command Syntax	1-8
Getting Help.	1-9
Help for a Command.	1-9
Help for a Topic.	1-10
2. Power Compiler Design Flow	
Power in the Design Cycle	2-2
Power Optimization and Analysis Flow	2-3
Simulation	2-5

Enable Power Optimization	2-5
Synthesis and Power Optimization	2-5
Power Analysis and Reporting	2-5
Power Compiler and Other Synopsys Tools	2-6
3. Power Modeling and Calculation	
Defining Power Types	3-2
Defining Static Power	3-2
Defining Dynamic Power	3-2
Switching Power	3-2
Internal Power	3-3
Calculating Power	3-4
Leakage Power Calculation	3-4
Multithreshold Voltage Libraries	3-6
Channel-width Based Leakage Power Calculation	3-6
Internal Power Calculation	3-8
NLDM Models	3-9
State and Path Dependency	3-11
Rise and Fall Power	3-12
Switching Power Calculation	3-12
Dynamic Power Calculation	3-13
Dynamic Power Unit Derivation	3-13
Multivoltage Power Calculation	3-14
Using CCS Power Libraries	3-16
4. Generating Switching Activity Information Format Files	
About Switching Activity	4-2
Introduction to SAIF Files	4-2
Generating SAIF Files	4-3
Generating SAIF Using VCD Output Files	4-4
Converting VCD file to a SAIF File	4-5
Limited SystemVerilog Support in vcd2saif Utility	4-5
Generating SAIF Files Directly From Simulation	4-6
Generating SAIF Files From SystemVerilog or Verilog Simulations	4-6
Generating SAIF File From RTL Simulation	4-6
Generating SAIF Files From Gate-Level Simulation	4-7

Understanding the VCS MX Toggle Commands.	4-8
Generating SAIF Files From VHDL Simulation	4-13
System Task List for SAIF File Generation from VHDL Simulation.	4-14
Verilog Switching Activity Examples	4-14
RTL Example	4-14
Verilog Design Description	4-15
RTL Testbench	4-16
RTL SAIF File	4-17
Gate-Level Example	4-18
Gate-Level Verilog Module	4-19
Verilog Testbench	4-19
Gate-Level SAIF File.	4-20
VHDL Switching Activity Example	4-22
VHDL Design Description	4-22
RTL Testbench.	4-22
RTL SAIF File	4-23
Analyzing a SAIF File	4-24
5. Annotating Switching Activity	
Switching Activity That You Can Annotate	5-2
Annotating Switching Activity Using RTL SAIF Files	5-2
Using the Name-Mapping Database	5-3
Integrating the RTL Annotation With PrimeTime PX	5-4
Annotating Switching Activity Using Gate-Level SAIF Files	5-4
Reading SAIF Files Using the read_saif Command	5-4
Reading SAIF Files Using the merge_saif Command	5-6
Annotating Switching Activity With the set_switching_activity Command.	5-7
Fully Annotating Versus Partially Annotating the Design	5-9
Analyzing the Switching Activity Annotation	5-10
Removing Switching Activity Annotation.	5-11
Estimating Nonannotated Switching Activity.	5-12
Annotating Design Nets With Default Switching Activity Values.	5-12
Propagating Switching Activity.	5-13

Deriving State- and Path-Dependent Switching Activity	5-13
6. Performing Power Analysis	
Overview	6-2
Identifying Power and Accuracy	6-2
Setting Accuracy Expectations	6-3
Switching Activity Annotation	6-3
Delay Model	6-4
Correlation	6-4
Clock Tree Buffers	6-5
Complex Cells	6-5
Performing Gate-Level Power Analysis	6-5
Using the report_power Command	6-6
Using the report_power_calculation Command	6-8
Analyzing Power With Partially Annotated Designs	6-8
Power Correlation	6-9
Performing Power Correlation	6-10
Power Correlation Script	6-10
Design Exploration Using Power Compiler	6-11
Examining Other dc_shell Commands for Power	6-12
Characterizing a Design for Power	6-13
Reporting the Power Attributes of Library Cells	6-14
Using a Script File	6-15
Power Reports	6-15
Power Report Summary	6-15
Net Power Report	6-17
Cell Power Report	6-18
Hierarchical Power Reports	6-19
Power Report For Interface Logic Model	6-20
7. Clock Gating	
Introduction to Clock Gating	7-3
Using Clock-Gating Conditions	7-6
Clock-Gating Conditions	7-6

Enable Condition	7-7
Setup Condition	7-7
Overriding Clock-Gating Conditions.	7-8
Inserting Clock Gates	7-9
Using the compile_ultra -gate_clock Command	7-9
Using the insert_clock_gating Command	7-9
Clock-Gate Insertion in Multivoltage Designs	7-10
Clock Gating Flows	7-10
Inserting Clock Gates in the RTL Design.	7-10
Inserting Clock Gates in Gate-Level Design	7-11
Power-Driven Clock Gating	7-12
Specifying Clock-Gate Latency.	7-14
The set_clock_gate_latency Command.	7-14
The set_clock_latency Command	7-16
Applying Clock-Gate Latency	7-17
Resetting Clock-Gate Latency	7-17
Comparison of the Clock-Gate Latency Specification Commands	7-18
Calculating the Clock Tree Delay From Clock-Gating Cell to Registers	7-19
Specifying Setup and Hold	7-19
Predicting the Impact of Clock Tree Synthesis.	7-22
Choosing a Value for Setup	7-23
Choosing a Value for Hold	7-24
Choosing Gating Logic	7-24
Choosing a Configuration for Gating Logic	7-24
Choosing a Simple Gating Cell by Name	7-29
Choosing a Simple Gating Cell and Library by Name	7-29
Choosing an Integrated Clock-Gating Cell.	7-30
Choosing an Integrated Cell by Functionality	7-30
Choosing an Integrated Cell by Name	7-30
Choosing a Generic Integrated Cell	7-31
Specifying a Subset of Integrated Clock Gates for Clock-Gate Insertion	7-31
Using Setup and Hold for Integrated Cells	7-31
Designating Simple Cells Exclusively for Clock Gating	7-32
Selecting Clock-Gating Style	7-33
Choosing a Specific Latch and Library	7-34

Choosing a Latch-Free Style	7-34
Improving Testability	7-36
Inserting a Control Point for Testability	7-36
Improving Observability With <code>test_mode</code>	7-39
Choosing a Depth for Observability Logic	7-41
Connecting the Test Ports Throughout the Hierarchy	7-41
Using the <code>insert_dft</code> Command	7-42
Default Clock-Gating Style	7-42
Modifying the Clock-Gating Structure	7-44
Changing a Clock-Gated Register to Another Clock-Gating Cell	7-44
Removing Clock Gating From the Design	7-44
Rewiring Clock Gating After Retiming	7-45
Integrated Clock-Gating Cells	7-46
Generic Integrated Clock-Gating Cells	7-46
Examples	7-47
Library-Level Requirements	7-48
Integrated Clock-Gating Cell Attributes	7-49
Pin Attributes	7-51
Timing Considerations	7-51
Propagating Clock Constraints	7-52
Ensuring Accuracy When Using Ideal Clocks	7-52
Sample Clock-Gating Script	7-52
Clock-Gating Naming Conventions	7-54
Sample Script for Naming Style	7-56
Sample Script Output Netlist	7-56
Keeping Clock-Gating Information in a Structural Netlist	7-57
Automatic Identification of Clock-Gating Cells	7-58
Explicit Identification of Clock-Gating Cells	7-58
Usage Flow With the <code>write_script</code> Command	7-59
Usage Flow With the <code>identify_clock_gating</code> Command	7-60
Replacing Clock-Gating Cells	7-61
Optimizing Clock-Gate Insertion	7-64
Multistage Clock Gating	7-65
Hierarchical Clock Gating	7-68

Enhanced Register-Based Clock Gating	7-70
Performing Clock-Gating on DesignWare Components	7-72
Reporting Command For Clock Gates and Clock Tree Power	7-72
The report_clock_gating Command.	7-73
8. Operand Isolation	
Operand Isolation Overview	8-2
Observable Don't Care Conditions	8-3
Power Compiler Operand Isolation Approach.	8-3
Automatic Versus User-Driven Operand Isolation Insertion	8-4
Automatic Versus Manual Operand Isolation Rollback.	8-4
Operand Isolation Methodology Flows	8-4
Two-Pass Approach (Recommended)	8-5
One-Pass Approach	8-7
Sample Scripts.	8-9
Commands and Variables Related to Operand Isolation	8-11
Using Operand Isolation	8-12
Specifying Operand Isolation Style and Selecting Insertion Mode	8-12
Controlling the Scope for Operand Isolation	8-12
Defining User Directives	8-14
Operand Isolation Rollback	8-15
Automatic Rollback Mechanism	8-15
Manual Rollback Mechanism	8-16
Sample Scripts for Operand Isolation Rollback	8-16
Operand Isolation Reporting	8-18
Interoperability	8-19
Operand Isolation and Clock Gating	8-19
Operand Isolation and Testability.	8-20
Debugging Tips.	8-21
Examples	8-22
Verilog RTL With Observable Don't Care Conditions	8-22
Report Operand Isolation Progress.	8-23
Examples of Using the set_operand_isolation_scope and set_operand_isolation_cell Commands	8-25
Operand Isolation Summary Report	8-27

9. Gate-Level Power Optimization

Overview	9-2
Input and Output of Power Optimization	9-2
Power Optimization in Synthesis Flow	9-4
General Gate-Level Power Optimization	9-5
Power Optimization Commands	9-5
Power Constraints	9-5
Scope of Power Constraints	9-6
Design Rule Constraints and Optimization Constraints	9-6
Cost Priority	9-6
Positive Timing Slack	9-7
Unmet Constraints	9-8
Design Rule Fixing	9-8
Incremental Optimization	9-8
Synthesizable Logic	9-9
Leakage Power Optimization	9-9
Enabling Leakage Optimization	9-9
Using Multithreshold Voltage Libraries	9-10
Library Threshold Voltage Attributes	9-10
Choosing the Leakage Power Calculation Model	9-12
Calculating Leakage Power	9-12
Sample Scripts For Leakage Optimization	9-13
Using the Default Usage Model	9-13
Using the Channel-Width Model	9-13
Power Critical Range	9-14
Dynamic Power Optimization	9-14
Running Dynamic Power Optimization	9-15
Annotating Switching Activity	9-15
Sample Scripts	9-15

10. Retention Register Implementation

Multithreshold-CMOS Retention Registers	10-2
Using Retention Registers	10-3
Library Level Requirements	10-4
Interaction With Other Synopsys Tools	10-5

11. Multivoltage Design Concepts

Multivoltage and Multisupply Designs	11-2
Preparing Libraries for Multivoltage Designs With Shut-Down Blocks	11-4
Using Target Library Subsets	11-6
Power Compiler Flows for Multivoltage Designs	11-7

12. IEEE 1801™ (UPF) Flow For Multivoltage Design Implementation

Multivoltage Design Concepts	12-2
Power Domains	12-2
Voltage Area	12-3
Level-Shifter and Isolation Cells	12-3
Always-on Logic Cells	12-4
Retention Register Cells	12-5
Basic Library Requirements for Multivoltage Designs	12-5
Power and Ground Pin Syntax	12-5
Target Library Subsetting	12-6
Synthesizing Multivoltage Designs Using UPF	12-7
Multivoltage Design Flow Using UPF	12-7
Defining Power Domains and Supply Network in UPF	12-10
Defining Power State Tables	12-16
Multivoltage Power Constraints	12-17
Defining Multivoltage Design Strategies	12-17
Defining the Level-Shifter Strategy	12-18
Associating Specific Library Cells With the Level-Shifter Strategy	12-20
Defining the Isolation Strategy	12-20
Associating Specific Library Cells With the Isolation Strategy	12-22
Defining the Retention Strategy	12-23
Mapping the Retention Registers	12-24
Handling Always-On Logic	12-25
Marking Pass-Gate Library Pins	12-25
Marking Leaf Cell Instance Pins As Always-On	12-25
Marking Library Cells for Always-On Optimization	12-26
Automatic Always-On Optimization	12-26

Performing Always-on Optimization on Top-level Feedthrough Nets	12-27
Identifying User-Defined Always-On Paths	12-27
Using Basic Gates as Isolation Cells	12-27
Additional Commands to Support Multivoltage Designs	12-28
create_voltage_area	12-28
hookup_retention_register	12-29
Reporting Commands in the UPF Flow	12-29
report_power_domain	12-29
report_level_shifter	12-29
report_power_switch	12-30
report_pst	12-30
report_isolation_cell	12-30
report_retention_cell	12-31
report_supply_net	12-31
report_supply_port	12-31
report_target_library_subset	12-31
check_mv_design	12-31
Hierarchical UPF Design Methodology	12-32
Guidelines	12-33
Methodology Guidelines	12-33
Interface Guidelines	12-33
Understanding the Hierarchical UPF Design Methodology	12-34
Block-Level Implementation	12-34
Top-Level Implementation	12-36
Assembling Your Design	12-37
13. Power Optimization in the Multivoltage Domain	
Clock-Gating Flow	13-2
Power-Gating Flow	13-2
Gate-Level Power Optimization Flow	13-3
Reporting	13-4
14. Multicorner-Multimode Optimization	
Basic Multicorner-Multimode Concepts	14-2
Scenario Definition	14-2

Multicorner-Multimode Optimization	14-2
Supported Features	14-2
Unsupported Features	14-3
Concurrent Multicorner-Multimode Optimization and Timing Analysis	14-3
Basic Multicorner-Multimode Flow	14-3
Setting Up the Design for a Multicorner-Multimode Flow	14-5
Specifying TLUPlus Files	14-5
Specifying Operating Conditions	14-6
Specifying Constraints	14-6
Handling Libraries in the Multicorner-Multimode Flow	14-6
Link Libraries With Equal Nominal PVT Values	14-7
Setting the dont_use Attribute on Library Cells in the Multicorner-Multimode Flow	14-9
Distinct PVT Requirements	14-9
Unsupported k-factors	14-10
Automatic Detection of Driving Cell Library	14-11
Relating the Minimum Library to the Maximum Library	14-11
Unique Identification of Libraries Based on File Names	14-12
Automatic Inference of Operating Conditions for Macro, Pad and Switch Cells	14-12
Scenario Management Commands	14-15
Power Optimization Techniques	14-16
Optimizing for Leakage Power	14-16
Optimizing for Dynamic Power	14-18
Reporting Commands	14-19
report_scenario Command	14-19
Reporting Commands That Support the -scenario Option	14-20
Commands That Report the Current Scenario	14-21
Reporting Examples	14-22
Supported SDC Commands	14-28
Multicorner-Multimode Script Example	14-29

Appendix A. Integrated Clock-Gating Cell Example

Library Description	A-2
Sample Schematics	A-4

Rising-Edge Latch-Based Integrated Cells	A-5
Rising-Edge Latch-Free Integrated Cells	A-7
Falling Edge Latch-Based Integrated Cells	A-8
Falling-Edge Latch-Free Integrated Cells	A-10

Appendix B. Attributes for Querying and Filtering

Derived Attribute Lists.	B-2
Usage Examples.	B-4

Appendix C. Implementing Multivoltage Designs Using RTL Isolation and Power Constructs

Using Power Domains.	C-3
Inferring Power Domains From an RTL Design	C-4
Creating Power Domains	C-4
Setting an Always-On Strategy	C-10
Optimizing Always-On Logic	C-11
Constraining the Boundary Nets of a Power Domain	C-11
Inserting Buffer-Type Level-Shifter Cells.	C-12
Inserting Isolation Cells and Enable-Type Level Shifters	C-16
Checking the Design for Level Shifters and Level-Shifter Violations	C-19
Removing Level Shifters	C-20
Power Gating	C-20
Power Gating Using Retention Registers	C-20
Use Models for Mapping	C-22
Power Gating Default Type	C-22
Based on the HDL Block Name	C-22
Named Blocks per Retention Register Type	C-24
Design-Level Mapping	C-24
Instance-Based Mapping	C-27
Specification of Which Registers Not to Map	C-27
Re-Mapping of Gate-level Designs to Retention Registers	C-27
Wiring Power-Gating Pins	C-28
Specifying Wiring Rules	C-28
Use Models for Wiring	C-30
Wiring by Power-Gating Cell Type	C-30
Wiring by Power-Gating Pin Name	C-31

Reporting Retention Registers	C-32
Reporting Multivoltage Designs	C-33
Enhancements to Report Commands Used With Multivoltage Designs	C-33
Reports Used With Multivoltage Designs	C-35
Top-Down Compile Flow	C-38
Handling Designs That Use Isolation Cells and Level Shifters, Including Enable-Type Level Shifters	C-40
Inserting Level Shifters When Enable-Type Level Shifter Library Cells Are Available	C-41
Inserting Level Shifters When Enable-Type Level Shifter Library Cells Are <i>Not</i> Available	C-42
Top-Down Compile Example Script	C-43
Bottom-Up Compile Flow	C-45
Automated Chip Synthesis Flow	C-46
Power Compiler Flows for Multivoltage Designs	C-47
Multivoltage Elements: \$isolate and \$power	C-49
Isolation Cells	C-49
Power Domains	C-50
Multivoltage Elements: isolate() and power()	C-51
Isolation Cells, Buffers, and Latches	C-52
Power Domains	C-54

Index

Preface

This preface includes the following sections:

- [What's New in This Release](#)
- [About This User Guide](#)
- [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 *Power Compiler Release Notes* in SolvNet.

To see the *Power Compiler Release Notes*,

1. Go to the Download Center on SolvNet located at the following address:

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

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. Click Power Compiler, and then select a release in the list that appears.

About This User Guide

This user guide describes the Power Compiler tool, its methodology, and its use. Power Compiler is a comprehensive tool that assists you in analysis and optimization of your design for power.

Audience

The *Power Compiler User Guide* builds on concepts introduced in Design Compiler publications. It is assumed in this user guide that the user has some familiarity with Design Compiler products.

Related Publications

For additional information about Power 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:

- Design Compiler
- DFT Compiler
- Formality
- PrimeTime PX

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
Courier	Indicates command syntax.
<i>Courier italic</i>	Indicates a user-defined value in Synopsys syntax, such as <i>object_name</i> . (A user-defined value that is not Synopsys syntax, such as a user-defined value in a Verilog or VHDL statement, is indicated by regular text font italic.)
Courier bold	Indicates user input—text you type verbatim—in Synopsys syntax and examples. (User input that is not Synopsys syntax, such as a user name or password you enter in a GUI, is indicated by regular text font bold.)
[]	Denotes optional parameters, such as <i>pin1 [pin2 ... pinN]</i>
	Indicates a choice among alternatives, such as <i>low medium high</i> (This example indicates that you can enter one of three possible values for an option: low, medium, or high.)
—	Connects terms that are read as a single term by the system, such as <i>set_annotated_delay</i>
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.
\	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 <https://solvnet.synopsys.com> (Synopsys user name and password required), then clicking “Enter a Call to the Support Center.”
- Send an e-mail message to your local support center.
 - E-mail support_center@synopsys.com from within North America.
 - Find other local support center e-mail addresses at <http://www.synopsys.com/Support/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 Power Compiler

This chapter describes the Power Compiler methodology and describes power library models and power analysis technology. In addition, it provides library and license requirements.

Power Compiler is part of Synopsys's Design Compiler synthesis family. It performs both RTL and gate-level power optimization and gate-level power analysis. By applying Power Compiler's various power reduction techniques, including clock-gating, operand isolation, multivoltage leakage power optimization, and gate-level power optimization, you can achieve power savings, and area and timing optimization in the front-end synthesis domain.

This chapter contains the following sections:

- [Power Compiler Methodology](#)
- [Power Library Models](#)
- [Power Analysis Technology](#)
- [Power Optimization Technology](#)
- [Working With Power Compiler](#)

Power Compiler Methodology

With the increasing popularity of portable-oriented applications, low-power designs have become crucial elements for product success. Most especially, the static power (leakage power) consumption concern is more pronounced when the technology is smaller than 90nm in the ultra-deep sub-micron domain.

Power Compiler provides the following optimization features to address the issues the IC designers are facing:

- Leakage power/static power optimization:
 - Multivoltage threshold power optimization
 - Power gating
- Dynamic power:
 - Clock-gating insertion techniques: discrete components, integrated clock gating, generic integrated clock gating
 - Operand isolation
 - Gate-level power optimization
- Multivoltage support:
 - Power Compiler provides a complete methodology for low-power designs.
- Power optimization technology:
 - The power optimization technology optimizes your design for power consumption. It computes average power consumption based on the activity of the nets in your design. You can perform the following types of power optimization of your design:
 - Register transfer level using clock gating and operand isolation techniques.
 - Gate level power optimization including leakage optimization using cell libraries with multivoltage threshold voltages.
 - Gate level dynamic power optimization, through simulation and back annotation of switching activity.
- Power analysis technology
 - The power analysis technology analyzes your design for power consumption. Working in conjunction with Design Compiler, Power Compiler provides simultaneous optimization for timing, power, and area. You can perform power analysis of your design at the
 - Register transfer level using RTL simulation

- Gate level using RTL or gate-level simulation

Power Library Models

The power library model analyzes leakage, switching, and internal power.

For more information about library modeling and characterization for power, see the Library Compiler documentation.

The Power Compiler gate-level power model supports the following features:

- Composite current source (CCS) library support
- Lookup tables based on output pin capacitance and input transition time
- Cells with multiple output pins
- State-dependent and path-dependent internal power
- Leakage power, including state-dependent and path-dependent internal power
- Separate specification of rise and fall power in the internal power group

In addition, you can use the Composite Current Source (CCS) power model. CCS models represent the physical circuit properties more closely than other models to the simulated data obtained during characterization with Spice. It is a current-based power model that contains the following features:

- One library format suitable for a wide range of applications, including power analysis and optimization
- Power analysis with much higher time resolution compared to NLPM
- Dynamic power characterized by current waveforms stored in the library. The charge can be derived from the current waveform
- Leakage power modeled as the actual leakage current. The leakage current does not artificially depend on the reference voltage, as is the case with leakage power. This facilitates voltage scaling
- Standard-cell and macro-cell modeling

Power Analysis Technology

Power Compiler analyzes your design for net switching power, cell internal power, and leakage power.

Power Compiler enables you to perform power analysis of your design at two different levels of abstraction:

- Gate-level power analysis uses switching activity from RTL or gate-level simulation or user-annotation.

When analyzing a gate-level design, Power Compiler requires a gate-level netlist and some form of switching activity for the netlist. Using steps described in this book, Power Compiler enables you to capture the switching activity of primary inputs, primary outputs, and outputs of sequential elements during RTL simulation. After you annotate the captured activity on design elements, Power Compiler propagates switching activity through the nonannotated portions of your design.

Using power analysis by way of switching activity from RTL simulation provides a much faster turnaround than analysis using switching activity by way of gate-level simulation.

If you require more accuracy during the later stages of design development, you can annotate some or all of the nets of your design with switching activity from full-timing gate-level simulation.

Power Compiler offers the following power analysis features:

- Performs gate-level power analysis.
- Analyzes net switching power, cell internal power, and leakage power.
- Accepts input as either user-defined switching activity, switching activity from RTL or gate-level simulation, or a combination of both. The default is vector-free.
- Propagates switching activity during power analysis to nonannotated nets.
- Supports sequential, hierarchical, gated clock, and multiple-clock designs.
- Supports RAM and I/O modeling using a detailed state-dependent and path-dependent power model.
- Performs power analysis in a single, integrated environment at multiple phases of the design process.
- Reports power at any level of hierarchy to enable quick debugging.
- Reports capability to validate your testbench.
- Supports interfaces to NC-Sim, MTI, VCS-MX, Scirroco, and Verilog-XL simulators for toggle data.

Synopsys also provides another gate level detail power analysis tool called PrimePower. PrimePower can help analyze peak power, glitch power and X-state power. It also has time based power waveform and supports special modes of operation. For more information, see the *PrimePower User Guide*.

Power Optimization Technology

You can optimize your design for power using the following capabilities:

- RTL clock gating
- Operand isolation
- Gate-level multivoltage and dynamic power optimization

RTL clock gating is the most effective power optimization feature provided by Power Compiler. This is a high-level optimization technique that can save a significant amount of power by adding clock gates to registers that are not always enabled and with synchronous load-enable or explicit feedback loops. This greatly reduces the power consumption of the design by reducing switching activity on the clock inputs to registers and eliminating the multiplexers. It also results in a lower area consumption for your design.

The operand isolation feature could significantly reduce the power dissipation of a datapath intensive design at the cost of a slight increase in area and delay. With operand isolation, the inputs of the datapath operators are held stable whenever the output is not used.

RTL clock gating and operand isolation optimize for dynamic power and can be applied simultaneously on a design.

When a gate-level power optimization constraint is set in the design, by default, Power Compiler performs optimization to meet the constraints for design rule checking, timing, power and area in that order of priority.

The Power Compiler gate-level power optimization solution offers the following features:

- Push-button user interface to reduce power consumption
- Multivoltage libraries for leakage optimization with short turnaround time
- Simultaneous optimization for timing, power, and area
- Optimization based on circuit activity, capacitance, and transition times
- Power analysis capability; optimizes with the same detailed power library model used in analysis
- Operates within Galaxy platform and is compatible with other Synopsys tools (Design Compiler, Floorplan Manager, Module Compiler, DFT Compiler, and Formality)

Working With Power Compiler

This section provides information about the basic requirements to analyze and optimize for power.

Library Requirements

Power Compiler uses technology libraries characterized for power. You can characterize your library with the following power features:

Internal Power

To optimize for dynamic power, Power Compiler requires libraries characterized for internal power. This is the minimum library requirement to characterize for power. This characteristic accounts for short-circuit power consumed internal to gates.

Leakage Power

To optimize for static power, Power Compiler requires libraries characterized for leakage power. This characteristic accounts for the power dissipated while the device is not in use. Power Compiler also supports multivoltage libraries.

State and Path Dependency

To optimize for varying modes of operation, Power Compiler requires libraries characterized for state-dependency. To optimize for varying power consumption based on various input to output paths, Power Compiler requires libraries characterized for path-dependency.

To capture state-dependent and path-dependent switching activity from simulation, library cells must have state- and path- dependent information in the lookup tables for internal power and pin capacitance. Synopsys Power Compiler uses state-dependent and path-dependent switching activity to compute state-dependent and path-dependent switching power.

If you are developing libraries to use with Synopsys power products, see the Library Compiler documentation. Power Compiler supports non-linear power models, scalable polynomial equation power models, and composite current source libraries.

Command-Line Interface

Power Compiler is accessible from the Design Compiler command-line interface if you have an appropriate license. See [“License Requirements” on page 1-7](#).

Using the Design Compiler command-line interface, power optimization takes place during your `dc_shell` optimization session. For more information about its command-line interface, see the Design Compiler documentation.

Power Compiler also works within the Design Compiler topographical domain shell (`dc_shell-topo`). Whereas `dc_shell` uses wide-load models for timing and area power optimizations, `dc_shell-topo` uses placement timing values instead. For more information, see the Design Compiler documentation.

Note:

Unless otherwise noted, all functionality described in this manual pertains to both `dc_shell` and `dc_shell-topo`. Also unless otherwise noted, this manual uses "`dc_shell`" as a generic term that applies to the Design Compiler topographical domain also.

License Requirements

Power analysis and optimization are performed in the following manner:

- Power analysis using Power Compiler
- Power optimization using Power Compiler

Power Compiler License

Power Compiler analysis and optimization require either one of the following two combinations of licenses:

- `Power-Optimization`
- `Power-Analysis + Power-Optimization-Upgrade`

Power Compiler is incorporated within Design Compiler. You need the license for Design Compiler in addition to the power licenses mentioned above.

These licenses also allow you to perform multivoltage power optimization and analysis.

How the Licenses Work

When you invoke `dc_shell`, no power license is checked out until you use a Power Compiler feature. When the Power Compiler feature is completed, your power license is released.

Reading and Writing Designs

When using `dc_shell`, you read designs from disk before working on them, make changes to them, and write them back to the disk.

Power Compiler can read or write a gate-level netlist in any of the formats shown in [Table 1-1](#).

Table 1-1 File Formats and Extensions

Format	Default extension	File type	Special license key required?
db	.db	Synopsys internal database format	No
ddc	.ddc	Synopsys Design Compiler database format (the default)	No
EDIF	.edif	Electronic Design Interchange Format	No
equation	.eqn	Synopsys equation format	No
LSI	.NET	LSI Logic Corporation netlist format	Yes
MENTOR	.neted	Mentor intermediate netlist format (see Synopsys <i>Mentor Interface Application Note</i>)	Yes
PLA	.pla	Berkeley (Espresso) PLA format	No
ST	.st	Synopsys state table format	No
TDL	.tdl	Tegas Design Language (TDL) netlist format	Yes
Verilog	.v	Hardware Description Language	Yes
VHDL	.vhd	VHSIC Hardware Description	Yes

Supported .db technology library formats are NLPM and CCS.

Command Syntax

Power Compiler provides the same shell language and links to external computer-aided engineering tools as Design Compiler.

You can use `dc_shell` commands in the following two ways:

- Type single commands interactively in the appropriate shell.

- Execute command scripts in the shell. Command scripts are text files of shell commands and might not require your interaction to continue or complete a given process. A script can start the shell, perform various processes on your design, save the changes by writing them to a file, and exit the shell.

Getting Help

In the `dc_shell` command line, you can display information about your screen about commands and topics.

Help for a Command

The syntax of any `dc_shell` command is displayed when you use the `-help` option after the command name. The `-help` option displays the possible options for a command.

Example

```
dc_shell> read_saif -help
Usage: read_saif      # read SAIF file
      -input <file_name>      (the input SAIF file name)
      [-instance_name <string>]
                              (the instance in the SAIF file
                              containing the switching activity)
      [-target_instance <instance>]
                              (the target instance that will be
                              annotated with the SAIF information)
      [-names_file <file_name>]
                              (the accumulated name changes file name)
      [-ignore <string>]      (the relative instance name whose
                              switching activity will be ignored)
      [-ignore_absolute <string>]
                              (the absolute instance name whose
                              switching activity will be ignored)
      [-exclude <file_name>] (the file name that contains one
                              or more -ignore options)
      [-exclude_absolute <file_name>]
                              (the file name that contains one
                              or more -ignore_absolute options)
      ...
```

Use the `man` command to display the entire man page for a command.

Example

```
dc_shell> man report_rtl_power
```

The man page contains syntax and other detailed information.

Help for a Topic

The `help` command displays information about a `dc_shell` command, variable, or variable group.

The following syntax displays the `help` command:

```
help [topic]
```

Here, *topic* is the name of a command, variable, or variable group. If a topic is not named, the `help` command displays its own man page.

The `help` command enables you to display the man pages interactively while you are running the shell. The online help includes man pages for all commands, variables, and variable groups.

The following example returns the man page for the `system_variables` variable group:

```
dc_shell> help system_variables
```

If you request help for a topic that cannot be found, Power Compiler displays the following error message:

```
dc_shell> help xyz_topic  
Error: No manual entry for 'xyz_topic'
```

2

Power Compiler Design Flow

As you create a design, it moves from a high level of abstraction to its final implementation at the gate level. Power Compiler offers analysis and optimization throughout the design cycle, from RTL to the gate level.

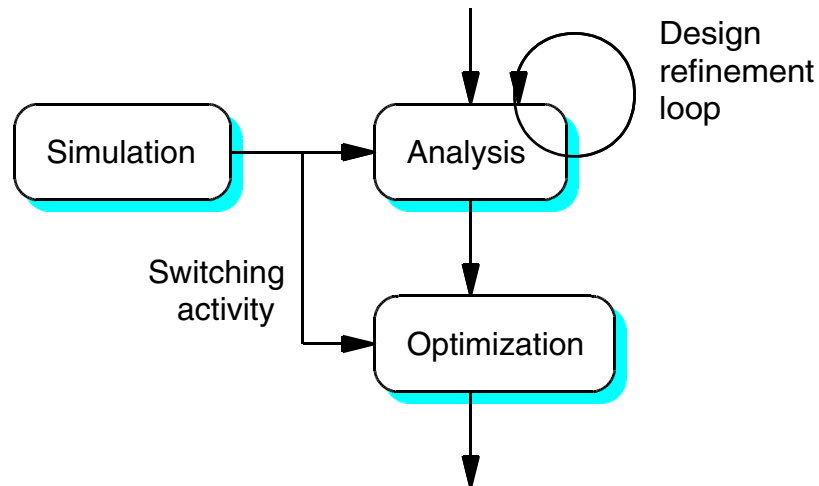
This chapter contains the following sections:

- [Power in the Design Cycle](#)
- [Power Optimization and Analysis Flow](#)
- [Power Compiler and Other Synopsys Tools](#)

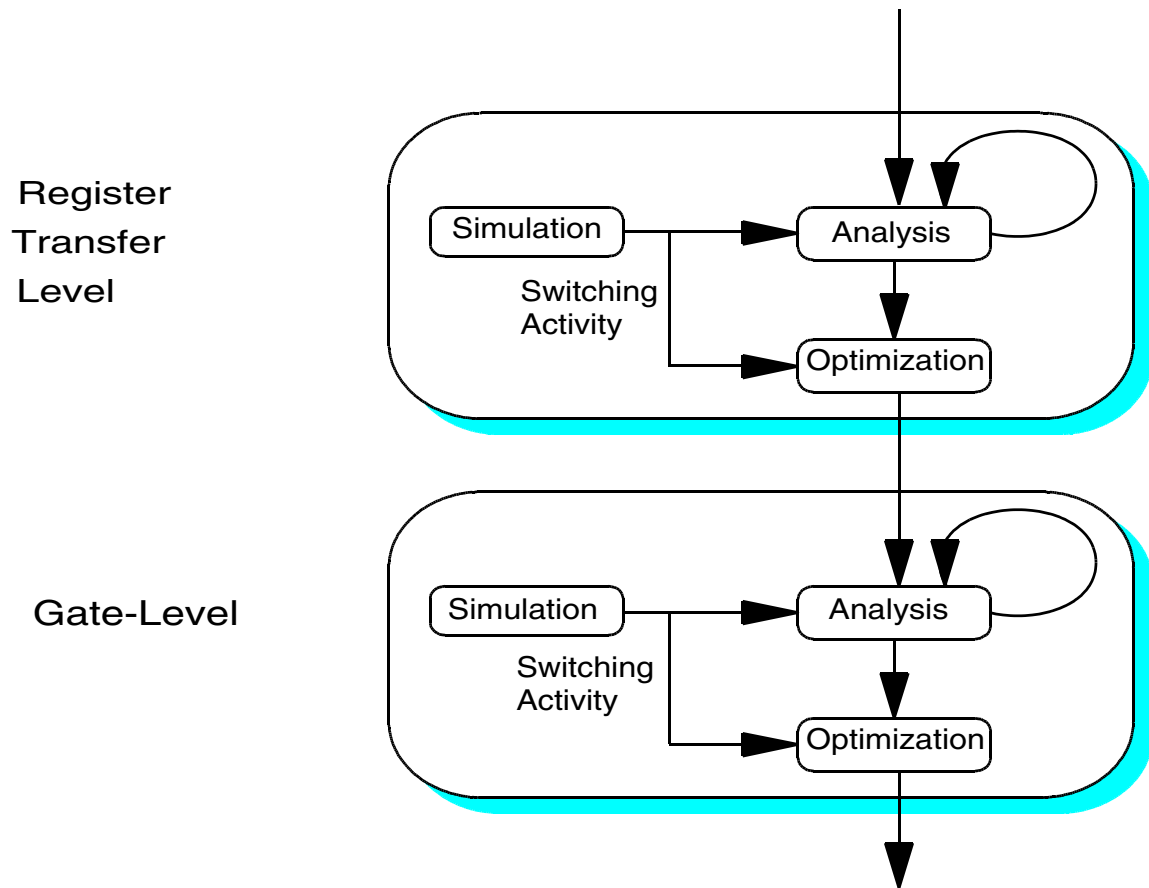
Power in the Design Cycle

At each level of abstraction, you use simulation, analysis, and optimization to refine your design before moving to the next lower level of design abstraction. The relationship of these three processes is shown in [Figure 2-1](#).

Figure 2-1 Power Flow at Each Abstraction Level



Simulation, analysis, and optimization occur at each level of abstraction. Design refinement loops occur within each level. Simulation and the resultant switching activity give analysis and optimization the necessary information to refine the design before going to the next lower level of abstraction. The entire flow is shown in [Figure 2-2](#).

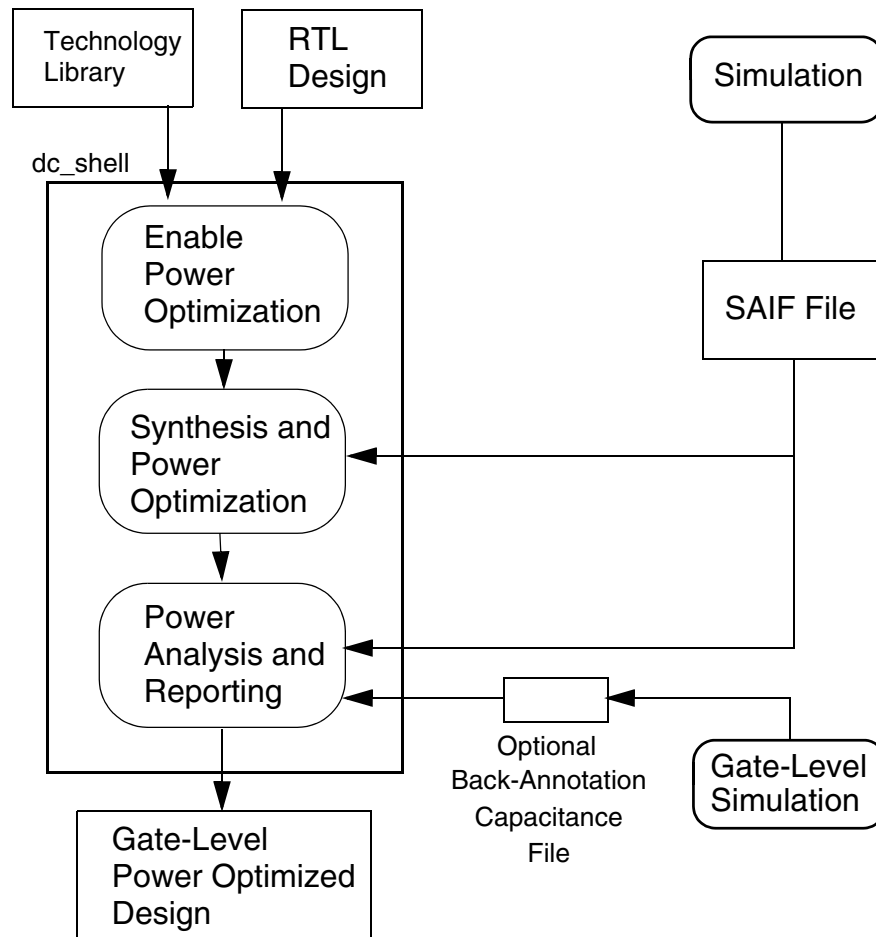
Figure 2-2 Power Flow From RTL to Gate-Level

Using Power Compiler, you can analyze and optimize at the RTL and gate levels. The higher the level of design abstraction, the greater the power savings you can achieve.

Power Optimization and Analysis Flow

Figure 2-3 shows a high-level power optimization and analysis flow.

Figure 2-3 Power Optimization and Analysis Flow



The power methodology starts with your RTL design and technology library and results in a power-optimized gate-level netlist.

During analysis and optimization, Synopsys power tools use information in your technology library. To optimize or analyze dynamic power and leakage power, your technology library must be characterized for internal power. To optimize or analyze static power, your technology library must be characterized for leakage power.

You can use Power Compiler to analyze the gate-level netlist produced by Design Compiler or the power-optimized netlist produced by Power Compiler.

Simulation

Most of the steps in the flow occur within the Design Compiler environment, `dc_shell`. However, [Figure 2-3](#) shows that the power flow requires a SAIF file, which is generated by simulation.

Simulation generates information about the design's switching activity and creates a Switching Activity Information Format (SAIF) file, which is used for annotation purposes. For information, see Chapter 4, "Generating Switching Activity Information Format Files."

During power analysis, Power Compiler uses annotated switching activity to evaluate the power consumption of your design. During power optimization, Power Compiler uses annotated switching activity to make optimization decisions about your design. For information, see Chapter 5, "Annotating Switching Activity."

Enable Power Optimization

Power Compiler provides several techniques for optimizing power, such as clock gating and operand isolation. Power optimization achieved at higher levels of abstraction has an increasingly important impact on reduction of power in the final gate-level implementation. You enable power optimizations with Power Compiler commands described in this manual. For information, see Chapter 7, "Clock Gating," and Chapter 8, "Operand Isolation."

Synthesis and Power Optimization

Design Compiler and Power Compiler work together within the `dc_shell` environment to synthesize your design to a gate-level netlist optimized for power. Synthesis with power optimization occurs during Design Compiler's compile processing.

Power Analysis and Reporting

You can use Power Compiler for analysis of your gate-level design at several points in your methodology flow. [Figure 2-3](#) shows power analysis after power optimization, which results in a detailed report of your power-optimized netlist.

You can also analyze power prior to synthesis and power optimization. For example, after annotating the switching activity from your SAIF file to verify that the annotation is correct. Analysis prior to power optimization provides an optional reference point for comparison with the power-optimized netlist.

Power Compiler and Other Synopsys Tools

Power Compiler enables you to use low-power methodology with the following Synopsys tools in addition to Design Compiler:

- DFT Compiler
- Formality

3

Power Modeling and Calculation

As you create a design, it moves from a high level of abstraction to its final implementation at the gate level. Power Compiler offers analysis and optimization throughout the design cycle, from RTL to the gate level.

This chapter contains the following sections:

- [Defining Power Types](#)
- [Calculating Power](#)
- [Using CCS Power Libraries](#)

Defining Power Types

The power dissipated in a circuit falls into two broad categories:

- Static power
- Dynamic power

Defining Static Power

Static power is the power dissipated by a gate when it is not switching, that is, when it is inactive or static.

Static power is dissipated in several ways. The largest percentage of static power results from source-to-drain subthreshold leakage, which is caused by reduced threshold voltages that prevent the gate from completely turning off. Static power is also dissipated when current leaks between the diffusion layers and the substrate. For this reason, static power is often called leakage power.

Defining Dynamic Power

Dynamic power is the power dissipated when the circuit is active. A circuit is active anytime the voltage on a net changes due to some stimulus applied to the circuit. Because voltage on an input net can change without necessarily resulting in a logic transition on the output, dynamic power can be dissipated even when an output net doesn't change its logic state.

The dynamic power of a circuit is composed of two kinds of power:

- Switching power
- Internal power

Switching Power

The switching power of a driving cell is the power dissipated by the charging and discharging of the load capacitance at the output of the cell. The total load capacitance at the output of a driving cell is the sum of the net and gate capacitances on the driving output.

Because such charging and discharging are the result of the logic transitions at the output of the cell, switching power increases as logic transitions increase. Therefore, the switching power of a cell is a function of both the total load capacitance at the cell output and the rate of logic transitions.

Internal Power

Internal power is any power dissipated within the boundary of a cell. During switching, a circuit dissipates internal power by the charging or discharging of any existing capacitances internal to the cell. Internal power includes power dissipated by a momentary short circuit between the P and N transistors of a gate, called short-circuit power.

To illustrate the cause of short-circuit power, consider the simple gate shown in [Figure 3-1](#). A rising signal is applied at IN. As the signal transitions from low to high, the N type transistor turns on and the P type transistor turns off. However, for a short time during signal transition, both the P and N type transistors can be on simultaneously. During this time, current I_{sc} flows from V_{dd} to GND, causing the dissipation of short-circuit power (P_{sc}).

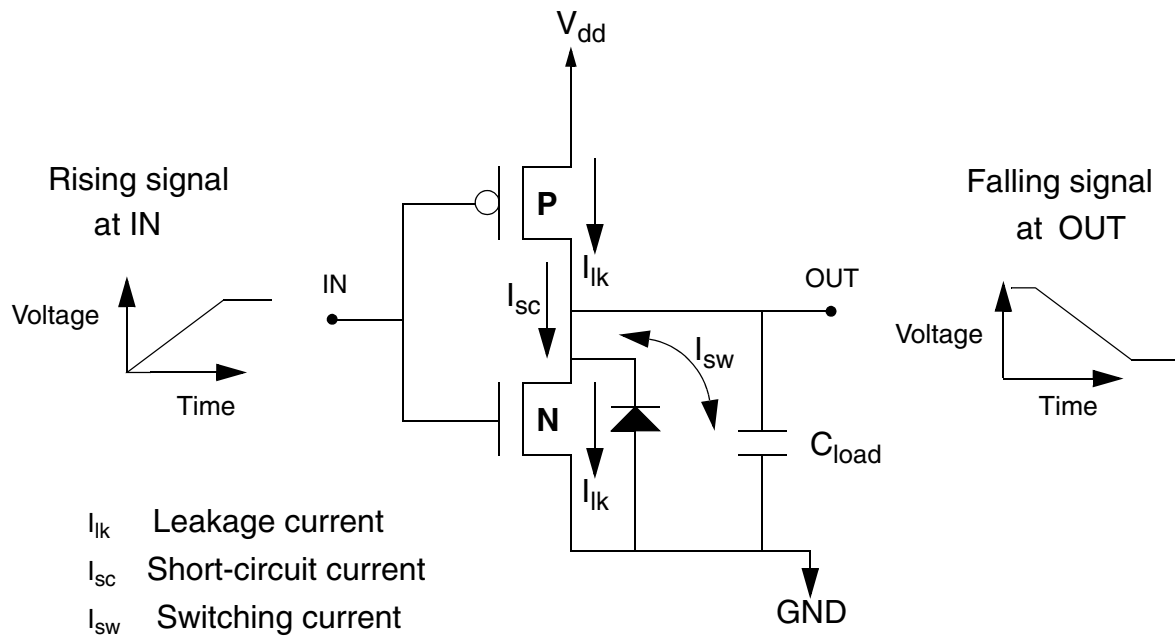
For circuits with fast transition times, short-circuit power can be small. However, for circuits with slow transition times, short-circuit power can account for 30 percent of the total power dissipated by the gate. Short-circuit power is affected by the dimensions of the transistors and the load capacitance at the gate's output.

In most simple library cells, internal power is due mostly to short-circuit power. For more complex cells, the charging and discharging of internal capacitance may be the dominant source of internal power.

Library developers can model internal power by using the internal power library group. For more information about modeling internal power, see the Library Compiler documentation.

[Figure 3-1](#) shows a simple gate and illustrates where static and dynamic power are dissipated.

Figure 3-1 Components of Power Dissipation



Calculating Power

Note:

The power calculations described in this section only apply to NLPM power calculations.

Power analysis calculates and reports power based on the equations that accompany this chapter. Power Compiler uses these equations and the information modeled in your technology library to evaluate the power of your design. This chapter includes information about library modeling for power where equations for power types appear. For more information about modeling power in your technology library, see the Library Compiler documentation.

Leakage Power Calculation

Power Compiler analysis computes the total leakage power of a design by summing the leakage power of the design's library cells, as shown in the following equation:

$$P_{\text{LeakageTotal}} = \sum_{\forall \text{cells}(i)} P_{\text{CellLeakage}_i}$$

Where:

$P_{\text{LeakageTotal}}$ = Total leakage power dissipation of the design

$P_{\text{CellLeakage}i}$ = Leakage power dissipation of each cell i

Library developers annotate the library cells with appropriate total leakage power dissipated by each library cell. They can provide a single leakage power for all cells in the library by using the `default_cell_leakage_power` attribute or provide leakage power per cell with the `cell_leakage_power` attribute.

If the `cell_leakage_power` attribute is missing or negative, the tool assigns the value of the `default_cell_leakage_power` attribute. If this is not available, Power Compiler assumes the default value of 0.

To model state-dependent leakage, use the `cell_leakage_power` attribute. You can also use boolean expressions to define the conditions for different cell leakage power values.

To calculate cell leakage, Power Compiler determines the units based on the `leakage_power_unit` attribute. It checks for the `leakage_power` attribute first. The leakage value for each state is multiplied by the percentage of the total simulation time at that state and summed to provide the total leakage power per cell.

If the state is not defined in the `leakage_power` attribute, the value of the `cell_leakage_power` attribute is used to obtain the contribution of the leakage power at the undefined state.

[Figure 3-2](#) shows the leakage power calculation performed on a NAND gate with state-dependent values.

Figure 3-2 Leakage Power Calculation for a NAND Gate With State Dependent Values



```
library ....
leakage_power_unit : 1nW ;
cell (NAND) ...
cell_leakage_power : .5 ;
leakage_power ( ) {
    when : "A&B"
    value : .2
```

For a total power consumption time of 600, the cell is at the state defined by the condition A&B for 33% of the time. For the remaining 67% of the simulation time, the default value is assumed.

Hence, the total cell leakage value is:

$$(.33 * .2nW) + (.67 * .5nW) = .4nW$$

Multithreshold Voltage Libraries

Static power dissipation has an exponential dependence on the switching threshold of the transistor's voltage. In order to address low-power designs IC foundries offer technologies that enable multiple threshold voltage libraries.

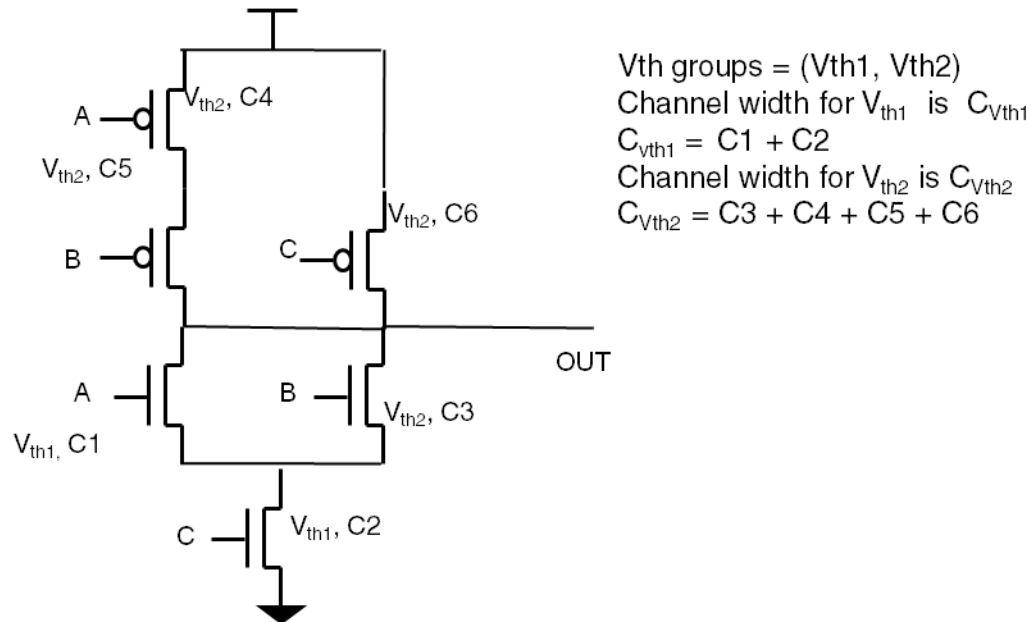
Each type of logic gate is available in two or more different threshold voltage (vth) groups. The threshold voltage determines the speed and the leakage characteristics of the cell. Cells with low-threshold transistors switch quickly but have higher leakage and consume more power. Cells with high threshold transistors have lower leakage and consume less power but switch more slowly.

For leakage power optimization Power Compiler supports multiple mechanisms for appropriately swapping high and low-threshold voltage cells based on the power and timing requirements.

Channel-width Based Leakage Power Calculation

The leakage power of a library cell is directly proportional to the channel-width of the transistors. In multithreshold libraries, cells with low-threshold voltage have faster timing and therefore can be used on the timing critical paths. Leakage power on these timing critical paths can be reduced by choosing lower voltage-threshold cells that have smaller channel-widths. The total channel-width for a specific threshold voltage group is obtained by summing the channel widths of all the transistors that belong to that threshold voltage group. [Figure 3-3](#) shows a CMOS cell with transistors from two different threshold voltage groups, vth1, vth2 and each threshold voltage group with different channel-widths, Cvth1 and Cvth2

Figure 3-3 CMOS Cell with Transistors From Two Threshold Voltage Groups



The cost function for the leakage power calculation is as follows:

$$\text{Min}(\sum W_{vth1} * C_{vth1} + \sum W_{vth2} * C_{vth2} + \dots)$$

Where:

vth1 and vth2 are the different threshold voltage groups

Cvth is the channel-width of the cell for the threshold voltage group vth

Wvth is the weight for the threshold voltage group vth.

This method of optimizing for leakage power is based only on the device dimension and is independent of the operating condition. For the tool to use the channel-width based leakage power optimization, the target library used must have the total channel widths of the transistors for each threshold voltage group for each cell.

The target libraries should have the cell-level and library-level attributes to specify the threshold voltage group and the corresponding channel-width values:

- The library level `threshold_voltage_groups` attribute and the corresponding channel-width attribute, `threshold_voltage_channel_width_factors`, should be mentioned in the library, as shown in the following example:

```
library (L1) {
  ...
```

```
threshold_voltage_groups(lvt, nvt, hvt);
threshold_voltage_channel_width_factors(100, 10, 1);
...
}
```

- The cell-level `threshold_voltage_groups` attribute and the corresponding channel-width attribute, `channel_widths`, should be mentioned in the library, as shown in the following example:

```
library (AN10) {
...
threshold_voltage_groups(lvt, nvt, hvt);
channel_widths(1.2, 12.5, 8.2);
...
}
```

- The `lc_enable_channel_width_based_leakage` variable must be set to true for the Library Compiler to recognize the channel-width related attributes.

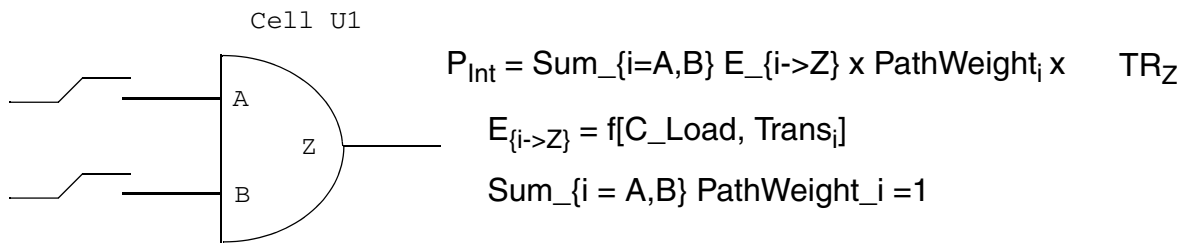
If the technology library is not characterized with the channel-width attribute, you can set these attributes in the Design Compiler script, using the `set_attribute` command. For more details on setting the attribute and an example script, see [“Sample Scripts For Leakage Optimization” on page 9-13](#).

Internal Power Calculation

When computing internal power, power analysis uses information characterized in the technology library. The `internal_power` library group and its associated attributes and groups define scaling factors and a default value for internal power. Library developers can use the internal power table to model internal power on any pin of the library cell.

A cell's internal power is the sum of the internal power of all of the cell's inputs and outputs as modeled in the technology library. [Figure 3-4](#) shows how Synopsys power tools calculate the internal power for a simple combinational cell, U1 with path-dependent internal power modeling.

Figure 3-4 Internal Power Model (Combinational)



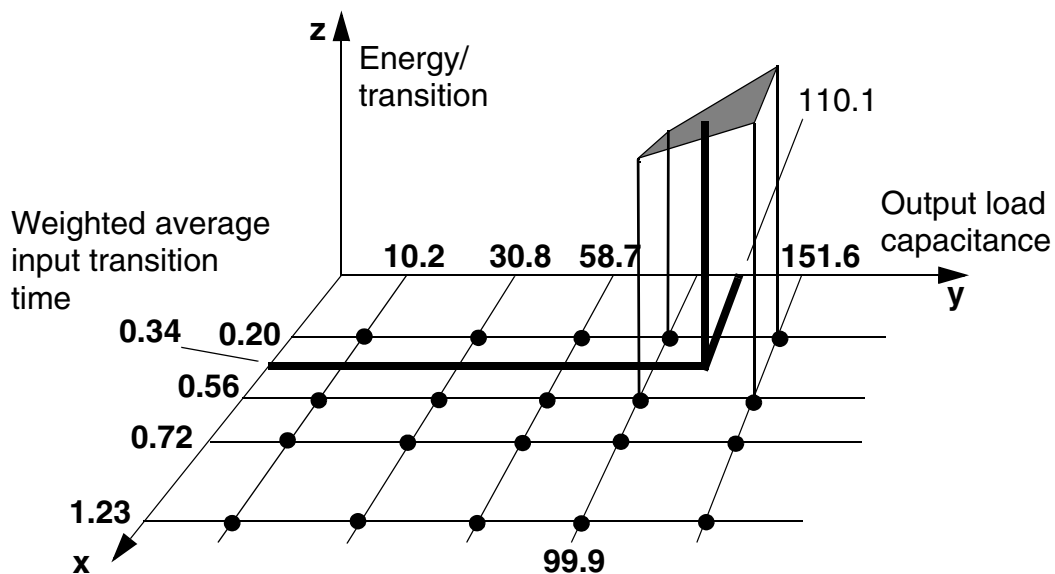
P_{Int}	Total internal power of the cell _E
E_Z	Internal energy for output Z as a function of input transitions, output load, and voltage
TR_Z	Toggle rate of output pin Z, transitions per second
TR_i	Toggle rate of input pin i, transitions per second
$Trans_i$	Transition time of input i
$\text{WeightAvg}_{(Trans)}$	Weighted average transition time for output Z

Power Compiler calculates the input path weights based on the input toggle rates, transition times, and functionality of the cell. Power Compiler supports NLDM (table based) models.

NLDM Models

To compute the internal power consumption of NLDM models, Power Compiler uses the weighted average transition time as an index to the internal power associated with the output pin. As an additional index to the power table, Power Compiler uses the output load capacitance. The two indexes enable Power Compiler to access the two-dimensional lookup table for the output, as shown in [Figure 3-5](#).

Figure 3-5 Two-Dimensional Lookup Table



For cells in which output pins have equal or opposite logic values, Power Compiler can use a three-dimensional lookup table. Power Compiler indexes the three-dimensional table by using input transition time and both output capacitances of the equal (or opposite) pins. The three-dimensional table is well suited to describing the flip-flop, which has Q and Q-bar outputs of opposite value.

The `internal_power` library group supports a one-, two-, or three-dimensional lookup table. Table 3-1 shows the types of lookup tables, whether they are appropriate to inputs or outputs, and how they are indexed.

Table 3-1 Lookup Tables

Lookup table	Defined on	Indexed by
One-dimensional	Input	Input transition
	Output	Output load capacitance
Two-dimensional	Output	Input transition and output load capacitance
Three-dimensional	Output	Input transition and output load capacitances of two outputs that have equal or opposite logic values

For more information about modeling internal power and library modeling syntax and methodology, see the Library Compiler documentation.

For various operating conditions, the table model supports scaling factors for the internal power calculation. These are listed below:

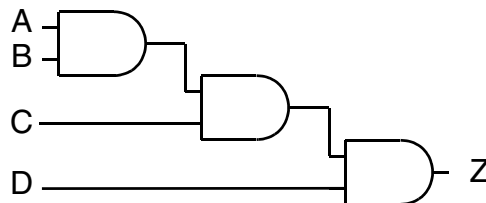
- `k_process_internal_power`
- `k_temp_internal_power`
- `k_volt_internal_power`

These factors however do not accurately model the non-linear effects of the operating conditions, so most vendors generate separate table-based libraries for different operating conditions.

State and Path Dependency

Cells often consume different amounts of internal power, depending on which input pin transitions or depending on the state of the cell. These are state and path dependent.

To demonstrate path-dependent internal power, consider the following simple library cell, which has several levels of logic and a number of input pins:



Input A and input D can each cause an output transition at Z. However, input D affects only one level of logic, whereas input A affects all three. An output transition at Z consumes more internal power when it results from an input transition at A than when it results from an input transition at D. You can specify multiple lookup tables for outputs, depending on the input transitions.

Power Compiler chooses the appropriate path dependent internal power table for an output by checking the `related_pin` attribute in the library. Based on the percentage of toggles on each input pin, the total power due to transitions on the output pin is calculated by accessing the correct table or equation for each related pin and applying the percentage contribution per input pin.

An example of a cell with state-dependent internal power is a RAM cell. It consumes a different amount of internal power depending on whether it is in read or write mode. You can specify separate tables or equations depending on the state or mode of the cell.

If the toggle rate information is provided for each state defined in the power model, Power Compiler accesses the appropriate information. If only the input/output toggle information is available, Power Compiler averages the tables for the different states to compute the internal power of the cell. For more information about how the toggle information affects the internal power analysis, see [Chapter 6, “Performing Power Analysis.”](#)

Rise and Fall Power

When a signal transitions, the internal power related to the rising transition is different from the internal power related to the falling transition. Power Compiler supports a library model that enables you to designate a separate rising and falling power value, depending on the transition.

Switching Power Calculation

Power Compiler analysis calculates switching power (P_c) in the following way:

$$P_c = \frac{V_{dd}^2}{2} \sum_{\forall \text{nets}(i)} (C_{Load_i} \times TR_i)$$

Where:

P_c Switching power of the design

TR_i Toggle rate of net i , transitions per second

V_{dd} Supply voltage

C_{Load_i} is the total capacitive load of net i , including parasitic capacitance, gate capacitance, and drain capacitance of all the pins connected to net i .

Power Compiler software obtains C_{Load_i} from the wire load model for the net and from the technology library information for the gates connected to the net. You can also back-annotate capacitance information after physical design.

Dynamic Power Calculation

Because dynamic power is the power dissipated when a circuit is active, the equations for switching power and internal power provide the dynamic power of the design.

Dynamic power = Switching power + Internal power

For more information about the library models, see the Library Compiler documentation.

Dynamic Power Unit Derivation

The unit for switching power and the values in the `internal_power` table is a derived unit. It is derived from the following function:

```
(capacitive_load_unit * voltage_unit2) / time_unit
```

The function's parameters are defined in the technology library. The result is scaled to the closest MKS unit: micro, nano, femto, or pico. This dynamic power unit scaling effect needs to be taken into account by library developers when generating energy values for the internal power table.

The following is an example of how Power Compiler derives dynamic power units (if the technology library has the following attributes):

```
capacitive_load_unit (0.35, ff);
voltage_unit: "1V"
time_unit: "1ns";
```

To obtain the dynamic power unit, complete the following steps:

1. Find the starting value.

```
starting value = capacitive_load_unit*voltage_unit2/
time_unit
starting value = .35e-15*(12)/1e-9
starting value = 3.5e-7W
```

The starting value consists of a base unit (1e-7W) and a multiplier (3.5).

2. Select an MKS base unit that converts the multiplier of the starting value found in step 1 to an integer number. For example, select an MKS unit between the range of att [1e-18] and giga [1e+12] watts, which converts the starting value's multiplier into an integer value.

The MKS base unit that meets this requirement in this example is nano [1e-9]. This is because the starting value of 3.5e-7W expressed in nW becomes 350nW. The original multiplier of 3.5 is converted to an integer value (350) by selecting the nW MKS base unit.

```
converted value = 350e-9W
```

```
converted value multiplier = 350
base unit = 1e9W = 1nW
```

3. Determine the base unit multiplier by selecting a power of 10 integer (for example, 1, 10, 100,...) closest in magnitude to the converted value multiplier found in step 2.

```
converted value multiplier = 350 (from step 2)
base unit multiplier = 100
```

4. Combine the base unit multiplier obtained in step 3 and the base unit obtained in step 2 to obtain the dynamic power unit.

```
base unit = 1nW (from step 2)
base unit multiplier = 100 (from step 3)
dynamic power unit = (100) 1nW = 100nW
```

In this example, each cell's dynamic power calculated by Power Compiler is multiplied by 100nW.

Multivoltage Power Calculation

Power Compiler supports the power analysis of libraries which contain cells with multiple rails for which power values are defined per voltage rail.

For multivoltage cells which contain separate power tables for each power level, Power Compiler correctly determines the internal and leakage power contribution for each power rail and sums it to report the total power consumption.

Shown below are sample cells which contain power tables per rail. For more information about defining per-rail power tables, see the Library Compiler documentation.

```
cell (AND2_1) {
  area : 1.0000;
  cell_footprint : MV12AND2;
  rail_connection (PV1, VDD1);
  rail_connection (PV2, VDD2);

  pin (a) {
    direction : input;
    capacitance : 0.1;
    input_signal_level : VDD1;
    internal_power () {
      power_level : VDD1;
      power (scalar) { values ( "1.0" ); }
    }
  }

  pin (b) {
    direction : input;
    capacitance : 0.1;
  }
}
```

```

    input_signal_level : VDD1;
    internal_power () {
        power_level : VDD1;
        power (scalar) { values ( "1.0" ); }
    }
}

pin (y) {
    direction : output;
    function : "a & b";
    output_signal_level : VDD2;

    timing () {
        related_pin : "a";
        timing_sense : positive_unate;
        cell_rise      ( scalar ) { values ( "1.0" ); }
        rise_transition ( scalar ) { values ( "1.0" ); }
        cell_fall      ( scalar ) { values ( "1.0" ); }
        fall_transition ( scalar ) { values ( "1.0" ); }
    }
    timing () {
        related_pin : "b";
        cell_rise      ( scalar ) { values ( "1.0" ); }
        rise_transition ( scalar ) { values ( "1.0" ); }
        cell_fall      ( scalar ) { values ( "1.0" ); }
        fall_transition ( scalar ) { values ( "1.0" ); }
    }
    internal_power () {
        power_level : VDD1;
        power (scalar) { values ( "1.0" ); }
    }
    internal_power () {
        power_level : VDD2;
        power (scalar) { values ( "2.0" ); }
    }
}
leakage_power () {
    power_level : VDD1;
    value : 1.0;
}
leakage_power () {
    power_level : VDD2;
    value : 2.0;
}
cell_leakage_power : 10;
}

```

Using CCS Power Libraries

CCS power libraries contain unified library data for power and rail analysis and optimization, which ensures consistent analysis and simplification of the analysis flow. By capturing current waveforms in the library, you can provide more accurate identification of potential problem areas.

Both CCS and NLPM data can co-exist in a cell description in the .lib file. That is, a cell description can have only NLPM data, only CCS data, or both NLPM and CCS data. Power Compiler uses either NLPM data or CCS data for the power calculation.

Use the `power_model_preference nlp | ccs` variable to specify your power model preference when the library contains both NLPM and CCS in it. The default value is `nlp`. Using CCS or NLPM power libraries does not change the use model.

For more information about CCS power libraries and how to generate them, see the Library Compiler documentation.

4

Generating Switching Activity Information Format Files

Power Compiler requires information about the switching activity of your design to perform power analysis and power optimization. You can use simulation tools such as VCS to generate switching activity information, for your design, either in VCD format or Switching Activity Information Format (SAIF). This chapter describes how to generate Switching activity in the SAIF.

This chapter contains the following sections:

- [About Switching Activity](#)
- [Introduction to SAIF Files](#)
- [Generating SAIF Files](#)
- [Verilog Switching Activity Examples](#)
- [VHDL Switching Activity Example](#)
- [Analyzing a SAIF File](#)

About Switching Activity

The dynamic component usually accounts for a large percentage of the total power consumption in a combinational circuit. Internal power of cells and transition from logic 1 to logic 0 and vice versa, directly affect the dynamic power of a design. This toggling of logic from one value to another is also known as switching activity.

Power Compiler models switching activity based on the following principles:

- Static Probability

Static probability is the probability that a signal is at a specific logic state; it is expressed as a number between 0 and 1. SP1 is the static probability that a signal is at logic-1. Similarly SP0 is that static probability that the signal is at logic-0.

You can calculate the static probability as a ratio of the time period for which the signal is at a certain logic state relative to the total simulation time. For example, if $SP1 = 0.70$, the signal is at logic 1 state 70 percent of the time. Synopsys power tools use SP1 when modeling switching activity.

- Toggle Rate

The toggle rate is the number of logic-0-to-logic-1 and logic-1-to-logic-0 transitions of a design object, such as a net, pin, or port, per unit of time. The toggle rate is denoted by TR.

When the switching activity information is available, you must annotate this information appropriately on the design objects, before you can use the switching activity information for power optimization and analysis. For more details on annotating switching activity see, [Chapter 5, "Annotating Switching Activity"](#).

Introduction to SAIF Files

The accuracy of the power calculations used by Power Compiler depends on the accuracy of the switching activity. Switching activity is calculated using RTL or gate-level simulation and is stored in a SAIF file. You can use the SAIF file to annotate switching activity information onto the design objects prior to power optimization and analysis.

SAIF is an ASCII format supported by Synopsys to facilitate the interchange of information between various Synopsys tools. You use the `read_saif` command to read SAIF file and the `write_saif` command to write out the SAIF file. For more information, see the man pages.

Early in your design cycle you can use RTL simulation to explore your design and find out,

- Which RTL architecture consumes the least power?

- Which module consumes the most power?
- Where is power being consumed within a given block?

Later in your design cycle, you can use the gate-level simulation rather than RTL simulation to annotate specific nets of your design or all the elements of your design for greater accuracy. [Table 4-1](#) summarizes the various methods of generating SAIF files and their accuracies.

Table 4-1 Comparing Methods of Capturing Switching Activity

Simulation	Captured	Not captured	Trade-offs
RTL	Synthesis-invariant elements	1. Internal nodes 2. Correlation of non-synthesis-invariant elements 3. Glitching 4. State and path dependencies	Fast runtime at expense of some accuracy
Zero-delay and unit-delay gate-level	1. Synthesis-invariant elements 2. Internal nodes 3. Correlation 4. State dependencies 5. Some path dependencies	1. Some path dependencies 2. Glitching	More accurate than RTL simulation, but significantly higher runtime
Full-timing gate-level	1. All elements of design 2. Correlation 3. State and path dependencies	Highest accuracy, but runtime can be very long	Correlation between primary inputs

Generating SAIF Files

You can generate a SAIF file either from RTL simulation or gate-level simulation. This section discusses both RTL and gate-level simulation using Synopsys VCS. VCS supports Verilog, SystemVerilog, and VHDL formats.

As shown in [Figure 4-1](#), you have two ways of generating a SAIF file:

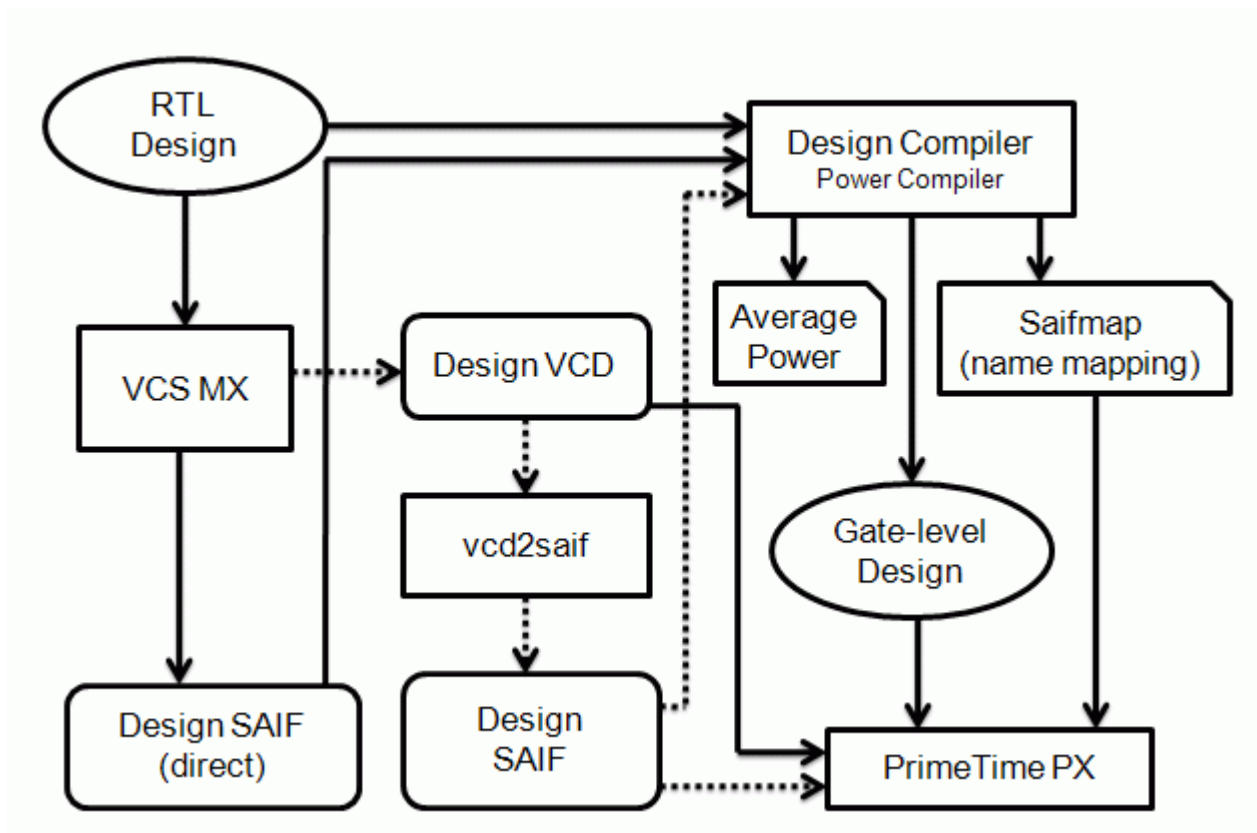
- The SAIF file can be generated directly by SAIF file.
- Alternatively, the SAIF file can be generated by using the vcd2saif utility to convert the VCD output file generated by VCS.

You can read the SAIF file into Power Compiler and generate a mapping file for all the name changes of the nodes. You then read the name-mapping file and the synthesized gate-level netlist in PrimeTime PX to perform averaged power analysis.

The solid lines indicate the recommended SAIF flow while the dotted lines indicate the alternate method of SAIF flow using various Synopsys tools.

The following sections discuss the various ways of generating the SAIF file.

Figure 4-1 SAIF File Generation and Its Usage With Various Synopsys Tools



Generating SAIF Using VCD Output Files

Using the VCD output files generated by VCS is the simplest method of generating SAIF files. The disadvantage is that VCD files can be very large, especially for gate-level simulation, requiring more time for processing.

Follow these steps to generate the SAIF file and to annotate the switching activity:

1. Run the simulation to generate VCD output file.

2. Use the `vcd2saif` utility to convert the VCD output file to a SAIF file.
3. Annotate the switching activity within the SAIF file as described in [Chapter 5, “Annotating Switching Activity.”](#)

Converting VCD file to a SAIF File

The `vcd2saif` utility converts the RTL or gate-level VCD file generated by VCS into a SAIF file. This utility has limited capability when the VCD is generated from the SystemVerilog simulation as described in [“Limited SystemVerilog Support in vcd2saif Utility” on page 4-5.](#)

The `vcd2saif` utility is architecture-specific and is located in `install_dir/$ARCH/syn/bin`. The `$ARCH` environment variable represents the specific platform (architecture) of your Synopsys software installation, such as linux, AMD.

You can use compressed VCD files (.Z) and gzipped VCD files (.gz). In addition, for VPD files, you can use the utility located at `$VCS_HOME/bin/vpd2vcd`, and for FSDB files, you can use the utility located at `$SYNOPSIS/bin/fsdb2vcd`.

You can use the following syntax for the `vcd2saif` utility for RTL simulation and gate-level simulation:

```
vcd2saif -i vcd_file -o bsaif
        [-instance path ...]
        [-format lang] [-testbench lang]
        [-verilog_instance path] [-vhdl_instance path]
        [-no_div] [-keep_leading_backslash] [-time]
```

The `vcd2saif` utility does not support state- and path- dependent switching activity. For information about each option, use the `vcd2saif -help`.

Limited SystemVerilog Support in vcd2saif Utility

The `vcd2saif` utility supports only a limited set of SystemVerilog constructs for VCD files that are generated from SystemVerilog simulation. [Table 4-2](#) shows the list of SystemVerilog constructs that are supported by the `vcd2saif` utility.

Table 4-2 SystemVerilog Constructs Supported by the vcd2saif Utility

System Verilog Constructs Supported by the vcd2saif Utility	
char	int
shortint	longint
bit	byte
logic	shortreal

Table 4-2 SystemVerilog Constructs Supported by the vcd2saif Utility

System Verilog Constructs Supported by the vcd2saif Utility	
void	enum
typedef	struct
union	arrays (packed and unpacked)

Generating SAIF Files Directly From Simulation

VCS MX can generate SAIF file directly from simulation. This direct SAIF file is smaller in size relative to the VCD files. Your input design for simulation can be a RTL or gate-level design. Also the design can be in Verilog, SystemVerilog, VHDL, or mixed HDL formats. When your design is in Verilog or SystemVerilog formats, you must specify system tasks to VCS MX using the toggle commands. If your design is in VHDL format, use the power command as described in [“Generating SAIF Files From VHDL Simulation” on page 4-13](#). For more details on the various supported formats and mixed language formats, see the *VCS MX/VCS MXi User Guide*.

The steps to follow to generate SAIF files for your designs are discussed in the following sections:

- [Generating SAIF Files From SystemVerilog or Verilog Simulations](#)
- [Generating SAIF Files From VHDL Simulation](#)

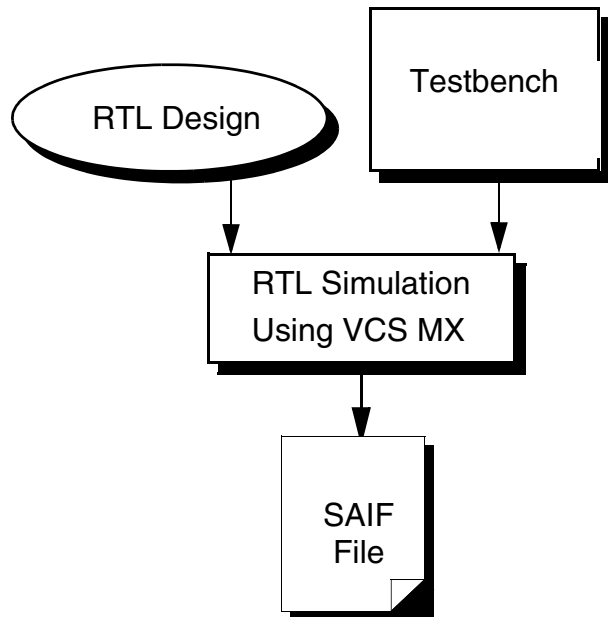
Generating SAIF Files From SystemVerilog or Verilog Simulations

Using VCS MX, you can generate SAIF files for your RTL as well as gate-level Verilog designs. When your design is in Verilog format, you must specify system tasks to VCS MX. These system tasks are also known as toggle commands. The system tasks specify the module for which switching activity is to be recorded and reported in the SAIF file. They also control the toggle monitoring during simulation.

Toggle commands are always preceded by the \$ symbol. For more details on toggle commands see, [“Understanding the VCS MX Toggle Commands” on page 4-8](#).

Generating SAIF File From RTL Simulation

[Figure 4-2](#) presents the methodology that you use to capture switching activity using RTL simulation. RTL simulation captures the switching activity of primary inputs, primary outputs, and other synthesis-invariant elements.

Figure 4-2 RTL Simulation using VCS MX

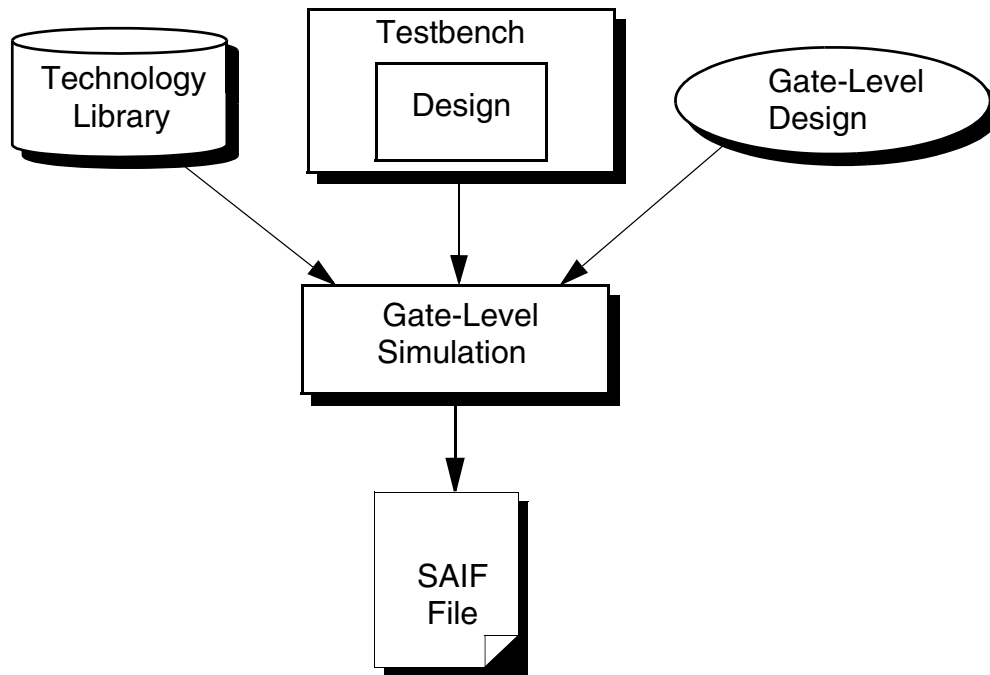
You follow these steps to capture switching activity using RTL simulation when your design is either in the Verilog or SystemVerilog format:

1. Specify the appropriate toggle commands in the testbench.
2. Run the simulation.

The SAIF file contains the switching activity information of the synthesis-invariant elements in your design. To use the SAIF file for synthesis using Power Compiler, annotate the switching activity in the SAIF file, as described in [Chapter 5, “Annotating Switching Activity.”](#)

Generating SAIF Files From Gate-Level Simulation

[Figure 4-3](#) presents the methodology that you use to capture switching activity using gate-level simulation. Gate-level simulation captures switching activities of pins, ports, and nets in your design.

Figure 4-3 Gate-Level Simulation using VCS MX

The steps that you follow to capture switching activity using gate-level simulation are similar to the steps that you follow for RTL simulation. These steps are to

1. Specify the appropriate toggle commands in the testbench.
2. Run the simulation.

The SAIF file contains information about the switching activity of the pins, ports, and nets in your design. It can represent the pin-switching activity, based on rise and fall values, if your technology library has separate rise and fall power tables.

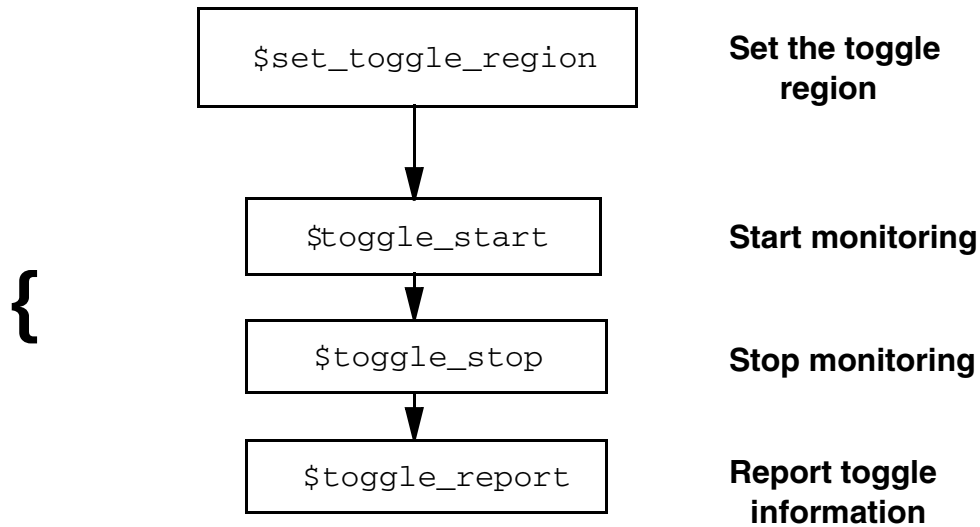
To use the SAIF file for synthesis using Power Compiler, annotate the switching activity in the SAIF file as described in [Chapter 5, “Annotating Switching Activity”](#).

Understanding the VCS MX Toggle Commands

When your design is in the Verilog or SystemVerilog format, to generate the SAIF file from RTL or from a gate-level simulation, you use the toggle commands to specify system tasks to VCS MX. The toggle commands start with the \$ symbol. Using the toggle commands, you can specify the subblock for toggle counting, defining specific periods for toggle counting during simulation. You can also control the start and stop of toggle counting.

Figure 4-4 presents an overview of the \$toggle commands in your testbench file. For simplicity, the figure omits optional commands.

Figure 4-4 Toggle Command Flow



The system level tasks that you specify to VCS MX, using the toggle commands are

1. Define the toggle region

Use the `$set_toggle_region` command to specify the toggle region. This command specifies the module instance for which the simulator records the switching activity in the generated SAIF file. The syntax of this command is as follows:

```
$set_toggle_region(instance [, instance]);
```

When you explicitly mention one or more module instance as the toggle region, simulator registers these objects and monitors them during simulation.

Note:

For gate-level simulation, if the technology library cell pins have rise and fall power values, their switching activity is monitored and reported for rise and fall separately.

2. Begin toggle monitoring

Use the `$toggle_start` command to instruct the simulator to start monitoring the switching activity. The syntax of this command is as follows:

```
$toggle_start();
```

During simulation, the tool starts monitoring the switching activities of the module instances that are defined in the toggle region. Toggle counting ignores the simulation activities that occur before the `$toggle_start` command.

Note:

The `$toggle_start` command does not take any parameters. You should define your toggle region before you start the toggle monitoring. This command monitors only the modules defined in the toggle region using the `$set_toggle_region` command.

3. End toggle monitoring

Use the `$toggle_stop` command to instruct the simulator to stop monitoring the switching activities.

During simulation, this command causes the simulation to stop monitoring the switching activities of the modules or instances in the toggle region.. Toggle counting and reporting ignore any simulation activity after the `$toggle_stop` command and before the `$toggle_start` command.

To use the `$toggle_stop` command, you must have already started the toggle counting using the `$toggle_start` command.

Note:

The `$toggle_stop` command does not take any parameters. This command causes the simulator to stop monitoring the switching activities for all the modules in the toggle region.

4. Report toggle information in an output file.

Use the `$toggle_report` command to write monitored gate and net switching activity to an output file. You can invoke `$toggle_report` any number of times using different parameters. For more details and examples of SAIF files, see [“RTL SAIF File” on page 4-17](#).

The syntax for the `$toggle_report` command is as follows:

```
$toggle_report (file_name,
               [synthesis_time_unit],
               instance_name_string,
               [hazard_rate, hazard_time]);
```

The values that you specify for the various options and parameters are as follows:

- ***file_name***

This is a required string parameter specifying the name you want for your switching activity output file. You can use any valid UNIX file name.

- ***synthesis_time_unit***

This optional parameter is the time unit of your synthesis library. Mention this time unit in seconds. For example, if the time unit in your synthesis library is 10 picoseconds, specify this value as 1.0e-11 for this parameter.

The `$toggle_report` command uses the number you pass to this parameter to convert simulation time units to synthesis time units. Power Compiler obtains the simulation time unit from simulation. If you don't specify the synthesis time unit parameter, a default value of 1 ns (1.0e-9) is used as the synthesis time unit.

- ***instance_name_string***

This required parameter is the full instance path name of the block from the top of your simulation environment down to the block or instance name that has the switching information you want in the output file. This parameter determines the hierarchy of the reported information in the output SAIF file.

Example

```
$toggle_report ("file.saif", 1.0e-11, "test.DUT");
```

In this example, the monitored design is DUT. The synthesis time unit is 1.0e-11. The instance name string is test. DUT and the output file is in SAIF (the default). The `strip_name_string` parameter is empty because SAIF accommodates the change in hierarchy between the simulation environment and the synthesis environment. Because `hazard_rate` is not passed, the software uses a default of 0.5. SAIF ignores the `hazard_time` parameter.

The `$toggle_report` command requires that you list parameters in the order shown in the syntax example.

Resetting the Toggle Counter

Use the `$toggle_reset` command to set the toggle counter to 0 for all the nets in the current toggle region. This command enables you to create different toggle monitoring periods in a simulation session.

For example, using `$toggle_start`, `$toggle_stop` or `$toggle_reset` with `$toggle_report`, you can create SAIF output files for specific periods during simulation. The syntax of this command is as follows:

```
$toggle_reset();
```

The `$toggle_reset` command has three requirements:

- You can invoke `$toggle_reset` only after you define a toggle region.
- You must invoke `$toggle_start`, `$toggle_stop`, and `$toggle_report`; otherwise, the command returns an error.
- You cannot pass parameters. This command sets the toggle count to 0 for all nets in the toggle region.

Capturing State- and Path-Dependent Switching Activity

By default, Power Compiler estimates the state- and path-dependent power information that is required for power calculations. However, if you want to obtain this information through simulation, you can use the `lib2saif` command prior to simulation. In this case, given a technology library, you can run the utility to obtain a library SAIF file that contains the state- and path-dependent information. This file is called the library forward-SAIF file. This file becomes the input to gate-level simulation.

The library forward-SAIF file contains information from the technology library about cells that have state and path dependencies. It can have rise and fall information if the library has separate rise and fall power tables.

To read the library forward-SAIF file into the simulator, use the `$read_lib_saif` command. This command registers the state- and path-dependent information for monitoring during simulation.

The syntax of the `$read_lib_saif` command is as follows:

```
$read_lib_saif(input_file);
```

For gate-level simulation, you must use the `$read_lib_saif` command to register state- and path-dependent cells and, by default, all internal nets in the design. The command registers state-dependent and path-dependent cells by reading the library forward-SAIF file. In addition, you must also set the toggle region for monitoring. If you do not use the `$read_lib_saif` command, the simulator registers all internal nets for monitoring by default.

You can use the `$read_lib_saif` command as often as you require during simulation; however, you must use this command before defining the toggle region using the `$set_toggle_region` command. When you define the toggle region, the `$set_toggle_region` command checks for the presence or absence of a `$read_lib_saif` command and registers internal nets accordingly.

Overriding Default Registration of Internal Nets

When you have the `read_lib_saif` command in the testbench, to override the default net monitoring behavior, use `$set_gate_level_monitoring` command to turn on or turn off the registration of internal nets.

The syntax for `$set_gate_level_monitoring` command is as follows:

```
$set_gate_level_monitoring ("on" | "off" | "rtl_on");
```


"on"

This string explicitly registers all internal nets for simulation. Thus, simulation monitors any internal net that is in the region defined using the `$set_toggle_region` command. Use double quotation marks as shown.

"off"

This string causes the simulator not to register any internal net. During simulation the tool does not monitor any internal net. Use double quotation marks as shown.

"rtl_on"

The registers in the toggle region are monitored while the nets in the toggle region are not monitored during simulation.

The `$set_gate_level_monitoring` command is optional. If you use it, you must do so before invoking the `$set_toggle_region`. After invoking the `$set_toggle_region` command, invoking the `$set_gate_level_monitoring` command causes an error, and simulation stops.

Generating SAIF Files From VHDL Simulation

You can use VCS MX to generate SAIF files from RTL or the gate-level simulation of VHDL designs. The methodology to generate the SAIF file is similar to the methodology used for Verilog designs, shown in [Figure 4-2](#) and [Figure 4-3](#). However you cannot use the toggle commands to specify the system tasks to the simulator.

For RTL-level VHDL files, variables are not supported by the simulator for monitoring. However, VHDL constructs such as generates, enumerated types, records, arrays of arrays are supported by VCS MX, for simulation.

The use model to generate a SAIF file from VHDL simulation consists of using the `power` command at the VCS MX command line interface, `simv`. The syntax of the `power` command is as follows:

```
power
  -enable
  -disable
  -reset
  -report file_name synthesis_time_unit scope
  -rtl_saif file_name
  [test_bench_path_name]
  -gate_level on| off | rtl_on
  region_signal_variable
```

- The `-enable` option enables the monitoring of the switching activity.
- The `-disable` option disables the monitoring of the switching activity.

- The `-reset` option resets the toggle counter
- The `-report` option reports the switching activity to an output file, SAIF file.
- The `-rtl_saif` option is used to read the RTL forward SAIF file.
- You can use `on`, `off` or `rtl_on` with the `-gate_level` option. [Table 4-3](#) summarizes the monitoring policy for VHDL simulation.

Table 4-3 Monitoring Policy for VHDL Simulation

Monitoring Policy	Ports	Signals	Variables
on	Yes	Yes	No
off	No	No	No
rtl_on	Yes	Yes	No

- You can specify either the toggle region and its children to be considered for monitoring, or the hierarchical path to the signal name.

System Task List for SAIF File Generation from VHDL Simulation

The following example script shows a sample task list that you specify to the simulator to generate a SAIF file. The design name is `test`. You can either specify each of these commands at the VCS MX command prompt or run the file that contains these commands.

```
power test
power -enable
run 10000
power -disable
power -report vhdl.saif 1e-09 test
quit
```

Verilog Switching Activity Examples

The following examples demonstrate RTL and gate-level descriptions with Verilog-generated switching activity data.

RTL Example

This Verilog RTL example includes the following elements:

- RTL design description
- RTL testbench

- SAIF output file from simulation

Verilog Design Description

[Example 4-1](#) shows the description for a state machine called foo.

Example 4-1 RTL Verilog Design Description

```
`timescale 1 ns / 1 ns

module foo ( data, clock, reset, dummy);

input [1:0] data;
input clock;
input reset;
output dummy;

wire dummy;

wire [1:0] NEXT_STATE;
reg [1:0] PRES_STATE;

parameter s0 = 2'b00;
parameter s5 = 2'b01;
parameter s10 = 2'b10;
parameter s15 = 2'b11;

function [2:0] fsm;
input [1:0] fsm_data;
input [1:0] fsm_PRES_STATE;

reg fsm_dummy;
reg [1:0] fsm_NEXT_STATE;

begin
  case (fsm_PRES_STATE)
    s0: //state = s0
      begin
        if (fsm_data == 2'b10)
          begin
            fsm_dummy = 1'b0;
            fsm_NEXT_STATE = s10;
          end
        else if (fsm_data == 2'b01)
          //....
        end
      end

    s5: //state = s5
      begin
        // ...
      end
  end
```

```
s10: //state = s10
begin
    // ...
end

s15: //state 15
begin
    // ...
end
endcase

    fsm = {fsm_dummy, fsm_NEXT_STATE};
end

endfunction

assign {dummy, NEXT_STATE} = fsm(data, PRES_STATE);

always @(posedge clock)
begin
    if (reset == 1'b1)
    begin
        PRES_STATE = s0;
    end
    else
    begin
        PRES_STATE= NEXT_STATE;
    end
end
endmodule
```

RTL Testbench

The Verilog testbench in [Example 4-2](#) simulates the design foo described in [Example 4-1](#). The testbench instantiates the design foo as U1.

Example 4-2 RTL Testbench

```

`timescale 1 ns / 1 ns

module stimulus;

  reg clock;
  reg [1:0] data;
  reg reset;
  wire dummy;
  foo U1 (data,clock, reset, dummy);

  always
  begin
    #10 clock = ~clock;
  end

  initial
  begin
    $set_toggle_region(stimulus.U1);
    $toggle_start();
    // ...
    clock = 0;
    data = 0;
    reset = 1;
    #50 reset = 0;
    #25 data = 3; #20 data = 0;
    #20 data = 1; #20 data = 2;
    // ...
    $toggle_stop();
    $toggle_report("my_rtl_saif", 1.0e-12, "stimulus");
    #80 $finish;
  end

```

RTL SAIF File

The RTL SAIF file is the output of RTL simulation and contains information about the switching activity of synthesis-invariant elements. The `$toggle_report` command creates this file.

[Example 4-3](#) is a SAIF file for the RTL Verilog cell description that is also shown in [Example 4-1 on page 4-15](#).

Example 4-3 RTL SAIF File

```

(SAIFFILE
(SAIFVERSION "2.0")
(DIRECTION "backward")
(DESIGN "foo")
(DATE "Mon May 11 18:54:04 2009")
(VENDOR "Synopsys, Inc")
(PROGRAM_NAME "Power Compiler")

```

```

(VERSION "3.0")
(DIVIDER / )
(TIMESCALE 1 ps)
(DURATION 1195000.00)
(INSTANCE stimulus
  (INSTANCE vendY
    (PORT
      (clock
        (T0 600000) (T1 595000) (TX 0)
        (TC 119) (IG 0)
      )
      (reset
        (T0 1145000) (T1 50000) (TX 0)
        (TC 1) (IG 0)
      )
      (dummy
        (T0 1085000) (T1 100000) (TX 10000)
        (TC 10) (IG 0)
      )
    )
  )
  (VIRTUAL_INSTANCE "sequential" data_reg[1]
    (PORT
      (Q
        (T0 995000) (T1 200000) (TX 0)
        (TC 12) (IG 0)
      )
    )
  )
  (VIRTUAL_INSTANCE "sequential" data_reg[0]
    (PORT
      (Q
        (T0 1035000) (T1 160000) (TX 0)
        (TC 7) (IG 0)
      )
    )
  )
)
)
)

```

Gate-Level Example

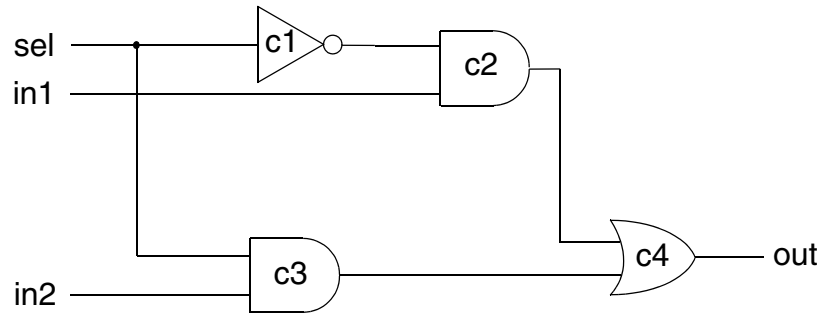
This Verilog gate-level example illustrates the following elements:

- Verilog cell description and schematic
- Verilog testbench
- SAIF output file from simulation

Gate-Level Verilog Module

Figure 4-5 shows the schematic for a simple multiplexer.

Figure 4-5 Schematic of Multiplexer Circuit: MUX21



Example 4-4 is the Verilog module that describes the MUX21 design.

Example 4-4 Verilog Module of Multiplexer Circuit: MUX21

```

/*`timescale 10ps/ 1ps
*/
module MUX21(out,d1,d2,sel);
input d1, d2, sel;
output out;
    IV c1(.Z(sel_),.A(sel));
    AN2 c2(.Z(d1m),.A(d1),.B(sel_));
    AN2 c3(.Z(d2m),.A(d2),.B(sel));
    OR2 c4(.Z(out),.A(d1m),.B(d2m));
endmodule
  
```

Verilog Testbench

The Verilog testbench in Example 4-5 tests the MUX21 design by simulating it and monitoring the various signals.

Example 4-5 Verilog Testbench for MUX21

```

/* Begin test.v */
`timescale 1ns/ 10ps
module top;
    reg in1, in2, sel;
    parameter hazrate = 0.99;
    parameter haztime = 0.23;

    MUX21 m1(out,in1,in2,sel);

    initial
    begin
        // start monitoring
        $monitor($time,,"in1=%b in2=%b sel=%b
  
```

```

out=%b",in1,in2,sel,out);

        // read SAIF file of state/path dependent info
        $read_lib_saif (cell.saif);

        // define the monitoring scope
        $set_toggle_region (m1);

        $toggle_start;

        // test first data line passing 0
        sel = 0;
        in1 = 0;
        in2 = 0;

        // test first data line passing 1
        #10 in1 = 1;

        #10 sel = 1;

        // test second data line passing 1
        #10 in2 = 1;

        $toggle_stop;
        $toggle_report("my_1st", 1.0e-9,"top.m1",
hazrate,
haztime,);

        // exit simulation
        $finish(2);
end
endmodule

```

The `$set_toggle_region` command sets the monitoring scope in module `m1` (the testbench instantiation of `MUX21`). All subsequent toggle commands affect only registered design objects and designs instantiated in registered objects. Thus, under `m1`, simulation monitors internal nets and state- and path-dependent cells (in this simple example, however, there are no subdesigns in `m1`).

The testbench example invokes `$toggle_report` command before exiting the simulation. Make sure that you declare any parameters you use for `$toggle_report` command in your testbench. These parameters appear at the top of the testbench in [Example 4-5 on page 4-19](#).

Gate-Level SAIF File

[Example 4-6](#) is an example of a SAIF file that results from gate-level simulation of `MUX21`.

Example 4-6 \$toggle_report Output File in SAIF

```

(SAIFFILE
(SAIFVERSION "2.0")
(DIRECTION "backward")
(DSIGN )
(DATE "Fri Oct 6 18:58:58 2000")
(VENDOR "Synopsys, Inc")
(PROGRAM_NAME "VCS-Scirocco-MX Power Compiler")
(VERSION "3.3")
(DIVIDER / )
(TIMESCALE 1 ns)
(DURATION 99999.00)
(INSTANCE tb
  (INSTANCE dut
    (NET
      (n12159
        (T0 99529) (T1 470) (TX 1)
        (TC 46) (IG 0)
      )
      (n12480
        (T0 0) (T1 99998) (TX 0)
        (TC 0) (IG 0)
      )
      (n12117
        (T0 61) (T1 99938) (TX 0)
        (TC 26) (IG 0)
      )
    )
  )
  (INSTANCE U12053
    (PORT
      (Z
        (T0 10) (T1 99989) (TX 0)
        (COND A (RISE)
          (IOPATH B (TC 0) (IG 0)
        )
        COND A (FALL)
          (IOPATH B (TC 0) (IG 0)
        )
        COND B (RISE)

```

```
(IOPATH A (TC 0) (IG 0)
)
COND B (FALL)
(IOPATH A (TC 1) (IG 0)
)
COND_DEFAULT (TC 1) (IG 0)
)
    )
        )
            )
```

This VHDL RTL example includes the following elements:

- RTL design description
- RTL testbench
- SAIF output file from simulation

Example 4-7 shows the description for a design called dummy .

```
library ieee;
use ieee.std_logic_1164.all;
entity dummy is
architecture beh of dummy is
    signal clk: std_logic := '0';
begin
    clk <= not clk after 5 ns;
end beh;
```

The RTL testbench in [Example 4-8](#) simulates the design foo described in [Example 4-7](#). The testbench instantiates the design dummy as dummy_ins.

Example 4-8 RTL Testbench

```

library ieee;
use ieee.std_logic_1164.all;
entity test is
end entity
architecture testbench of test is
  component dummy is
  end component;
begin
  dummy_ins: dummy;
end testbench;

```

RTL SAIF File

This RTL SAIF file is the output of RTL simulation and contains information about the switching activity of synthesis-invariant elements. The `power -report` command creates this file.

[Example 4-9](#) is a SAIF file for the RTL VHDL description that is shown in [Example 4-7](#).

Example 4-9 RTL SAIF File

```

/** There is no explicit set_gate_level_monitoring command, **/
/** and the default behavior is to monitor internal nets **/
(SAIFFILE
(SAIFVERSION "2.0")
(DIRECTION "backward")
(DESIGN )
(DATE "Tue May 5 05:56:35 2009")
(VENDOR "Synopsys, Inc")
(PROGRAM_NAME "VCS-Scirocco-MX Power Compiler")
(VERSION "1.0")
(DIVIDER / )
(TIMESCALE 1 ns)
(DURATION 10000.00)
(INSTANCE TEST
  (INSTANCE DUMMY_INS
    (NET
      (CLK
        (T0 5000) (T1 5000) (TX 0)
        (TC 1999) (IG 0)
      )
    )
  )
)
)
)

```

Analyzing a SAIF File

This section describes the various elements of SAIF files. For a sample SAIF file see [Example 4-3](#) and [Example 4-6](#). The definitions for various terminologies in the SAIF file are summarized in [Table 4-4](#).

Table 4-4 Definitions of Terminologies in the SAIF File

T0	Duration of time found in logic 0 state.
T1	Duration of time found in logic 1 state.
TX	Duration of time found in unknown “X” state.
TC	The sum of the rise (0 $\rightarrow$ 1) and fall (1 $\rightarrow$ 0) transitions that are captured during monitoring.
IG	Number of 0 $\rightarrow$ X $\rightarrow$ 0 and 1 $\rightarrow$ X $\rightarrow$ 1 glitches captured during monitoring.
RISE	Rise transitions in a given state.
FALL	Fall transitions in a given state.

Duration refers to the time span between `$toggle_start` and `$toggle_stop` in the testbench during simulation. During this time span, ports, pins, and nets are monitored for toggle activity. Use these definitions when analyzing a SAIF file.

5

Annotating Switching Activity

Switching activity is required for accurate power calculations. This chapter explains the different types of switching activity information and illustrates how you can annotate gate-level design objects with switching activity.

This chapter contains the following sections:

- [Switching Activity That You Can Annotate](#)
- [Annotating Switching Activity Using RTL SAIF Files](#)
- [Annotating Switching Activity Using Gate-Level SAIF Files](#)
- [Annotating Switching Activity With the `set\_switching\_activity` Command](#)
- [Fully Annotating Versus Partially Annotating the Design](#)
- [Analyzing the Switching Activity Annotation](#)
- [Removing Switching Activity Annotation](#)
- [Estimating Nonannotated Switching Activity](#)

Switching Activity That You Can Annotate

The power of a design depends on the switching activity of the design nets and cell pins, which must be annotated onto design objects like nets, ports, pins, and cells for use by the `report_power` command during power calculation.

The following types of switching activity can be annotated on design objects:

- Simple switching activity on design nets, ports and cell pins. Simple switching activity consists of the static probability and the toggle rate. The static probability is the probability that the value of the design object has logic value 1. The toggle rate is the rate at which the design object switches between logic values 0 and 1.
- State dependent toggle rates on input pins of leaf cells. As explained in [Chapter 3, “Power Modeling and Calculation,”](#) the internal power characterization of an input pin of a library cell can be state dependent. The input pins of instances of such cells can be annotated with state dependent toggle rates.
- State-dependent and/or path-dependent toggle rates on output pins of leaf cells. As explained in [Chapter 3, “Power Modeling and Calculation,”](#) the internal power characterization of output pins can be state dependent and/or path dependent. Output pins of cells with state- and path-dependent characterization can be annotated with state- and path-dependent toggle rates.
- State dependent static probability on leaf cells. Cell leakage power can be characterized using state dependent leakage power tables (see [Chapter 3, “Power Modeling and Calculation”](#)). Such cells can be annotated with state-dependent static probability.

Annotating Switching Activity Using RTL SAIF Files

Optimal power analysis and optimization results occur when switching activities reported in the RTL SAIF file are accurately associated with the correct design objects in the gate-level netlist. For this to occur, the RTL names must map correctly to their gate-level counterparts. During synthesis, however, mapping inaccuracies can occur that can affect your annotation.

To ensure proper name mapping and annotation for RTL SAIF files, do the following:

1. At the beginning of synthesis, specify the `saif_map -start` command.

This command causes Power Compiler to create a name-mapping database during synthesis optimization that Power Compiler then uses for power analysis and optimization.

2. After compile, specify `read_saif -auto_map_name` to perform RTL SAIF annotation using the name-mapping database.

If you plan to perform power optimization techniques that depend on switching activity, such as power-driven clock gating or operand isolation, specify this command prior to compile.

Using the Name-Mapping Database

You can access the name-mapping database on the rare occasion that the `read_saif -auto_map_name` annotation requires adjustment. Various `saif_map` options allow you to query, report, modify, save, clear, and load the database. Using the `-input` option of the `saif_map` command you can read a regular, uncompressed file or a compressed file in gzip format with a `.gzip` extension to the input file.

The command syntax is

```
saif_map [-start] [-end]
        [-reset] [-report] [-get_name]
        [-set_name name_list]
        [-add_name name_list]
        [-remove_name name_list]
        [-clear_name]
        [-get_object_names name_list]
        [-create_map]
        [-write_map file_name]
        [-read_map file_name]
        [-type type]
        [-inverted] [-instance objects]
        [-hierarchical] [-no_hierarchical]
        [-columns columns] [-sort columns]
        [-rtl_summary] [-missing_rtl]
        [-input SAIF_file] [-review] [-preview]
        [-source_instance SAIF_instance_name]
        [-target_instance target_instance_name]
        [-hsep character] [object_list]
        [-verbose] [-non_verbose] [-nosplit]
```

Suppose that after `read_saif -auto_map_name`, you want to review the name-mapping database and manually add a mapping entry yourself. The series of commands might look like this:

```
read_saif -auto_map_names -input ../sim/rtl.saif \
        -instance tb/dut -verbose
report_saif -hier -rtl -missing

reset_switching_activity

saif_map -add_name "Ax_ins" [get_port AX_usr_ins]
read_saif -auto_map_names -input rtl.saif ../sim/rtl/rtl.saif \
        -instance tb/dut
```

This example manually maps the RTL SAIF object “Ax_ins” and the design object “AX_use_ins.” The `read_saif -auto_map_names` command tells Power Compiler to perform annotation again with the modified database.

For information about the command options, see the man page.

Integrating the RTL Annotation With PrimeTime PX

Similar to Power Compiler, PrimeTime PX requires accurate RTL-to-gate name-mapping correspondence to perform accurate power analysis. Use Power Compiler to output the name-mapping files that PrimeTime PX can use for RTL-to-gate name mapping.

After `read_saif`, specify the `saif_map` command as follows to generate a name-mapping file that can be read directly into PrimeTime PX:

```
saif_map -type ptpx -write_map file_name
```

The name-mapping output file appears as follows:

```
set_rtl_to_gate_name -rtl{clk_sn} -gate clk_sn
set_rtl_to_gate_name -rtl{rx_top/data_i[9]} \
    -gate rx_top_data_i_reg<9>
...
```

Annotating Switching Activity Using Gate-Level SAIF Files

You can use either the `read_saif` or the `merge_saif` command to annotate switching activity.

Reading SAIF Files Using the `read_saif` Command

To annotate gate-level switching activity onto the gate-level netlist, use the `read_saif` command. For example,

```
dc_shell> read_saif -input file -instance TEST/DUT/U1
```

In this example, the `read_saif` command annotates the information in `file` onto the current gate-level design, `U1`. The `-instance` option identifies the hierarchical location of the current design in the simulation environment.

The input file specified using the `-input` option of the `read_saif` command can be a text file or a compressed gzip file with a `.gzip` extension. For example,

```
dc_shell> read_saif -input file.gzip -instance TEST/DUT/U1
```


A SAIF file is usually generated using an HDL simulation flow, where a simulation testbench instantiates the design being simulated and provides simulation vectors. The generated SAIF file contains the switching activity information organized in a hierarchical fashion, where the hierarchy of the SAIF file reflects the hierarchy of the simulation testbench. If a design is instantiated in the testbench (tb) as the instance i, then the SAIF file contains the switching activity information for the design under the hierarchy tb/i. In this case, the instance name `tb/i` should be used as the option to the `-instance` option when reading the SAIF file.

```
dc_shell> read_saif -input des.saif -instance tb/i
```

Specifying an invalid instance name results in having all or most of the switching activity stored in the SAIF file not read properly. An error message is printed if none of the information stored in the SAIF file is read by the `read_saif` command.

The SAIF file contains time duration values and specifies a time unit which is usually the time unit used during simulation. When reading the SAIF file, the `read_saif` command automatically converts the SAIF time units to the synthesis time units. The synthesis time units are obtained from the time units of the target or link library. When the synthesis time units cannot be obtained, the `read_saif` command prints a warning message and uses a default time unit of 1 ns. In such cases, the `-scale` and `-unit` options can be used to specify the intended synthesis time unit. For example, if a target technology library with the time units 100 ps is used for synthesis and a SAIF file is being read before the technology library is used (for linking or synthesis), you would use the options as follows:

```
read_saif -scale 100 -unit ps
```

When reading the SAIF file, the `report_lib` command gives the time units specified in a technology library. The `report_power` command gives the synthesis library time units used during power calculations.

This is the syntax of the command:

```
read_saif
    -input file_name
    [-instance_name string]
    [-target_instance instance]
    [-names_file file_name]
    [-ignore string]
    [-ignore_absolute string]
    [-exclude file_name]
    [-exclude_absolute file_name]
    [-scale scale_value]
    [-unit_base time_unit]
    [-khrate float]
    [-rtl_direct]
    [-verbose]
```

For information about the command options, see the man page for the `read_saif` command.

Reading SAIF Files Using the `merge_saif` Command

The `merge_saif` command can be used to read switching activity information from multiple SAIF files. Input SAIF files are given individual weights, and a weighted sum of the switching activities is annotated. This command can be used in flows where different SAIF files are generated for different modes of the same design. The switching activity from all the different modes can then be used for power calculations and optimization.

The following is an example of how `merge_saif` can be used. We assume that the design has three modes: standby, slow and fast; and that the SAIF files, `standby.bsaif`, `slow.bsaif` and `fast.bsaif` are generated for these modes. Depending on the expected usage of the design, we give the following weighting to each SAIF file:

standby.saif: 80%; slow.bsaif: 5%; fast.bsaif: 15%

The SAIF files can then be read using the following command:

```
dc_shell> merge_saif -input_list \  
               {-input standby.saif -weight 80 \  
                -input slow.bsaif   -weight 5  \  
                -input fast.bsaif   -weight 15 } \  
               -instance tb/i
```

When the output file specified, using the `-output` option, has a `.gzip` extension, the file written out is in the compressed gzip format. A regular, uncompressed file can also be written out using the `-output` option.

The `-output` option of the `merge_saif` command can be used to generate a SAIF file containing the weighted sum of the switching activities.

After the `merge_saif` command reads each individual SAIF file, it uses a switching activity propagation mechanism to estimate the switching activity of design nets that are not included in the SAIF file. You can therefore use the following to generate a gate-level SAIF file with estimated switching activity information from an RTL SAIF file:

```
dc_shell> merge_saif -input_list {-input rtl.bsaif -weight 100} \  
               -instance tb/i -output estimate.bsaif
```

The `-simple_merge` option can be used to switch off the switching activity propagation mechanism when the information in the SAIF files is being merged.

The syntax of the `merge_saif` command is the same as that of the `read_saif` command with the following exceptions:

- A weighted input file list is specified instead of a single input file
- The two options `-simple_merge` and `-output` can be used with `merge_saif`

This is the command syntax:

```
merge_saif
  -input_list weighted_filename_list
  [-simple_merge]
  [-output merged_saif_filename]
  [-instance_name string]
  [-scale scale_value]
  [-unit_base time_unit]
  [-strip_module string]
  [-ignore string]
  [-ignore_absolute string]
  [-exclude filename]
  [-exclude_absolute filename]
  [-rtl_direct]
  [-khrate float]
```

For more information, see the man page.

Annotating Switching Activity With the `set_switching_activity` Command

Use the `set_switching_activity` command to annotate particular design objects.

Note:

The `-toggle_rate` option differs from the toggle rate (TR) used for modeling switching activity, which is the number of logic transitions per unit of time. The `-toggle_rate` option expresses the sum of the rise and fall transitions that the signal makes during an entire simulation, clock period, or other period you specify. Power Compiler uses the `-toggle_rate` and `-period` (or `-clock`) options to determine the actual toggle rate of design objects.

Annotate simple switching activity using the `-static_probability`, `-toggle_rate` and `-period` options. For example:

```
set_switching_activity [get_net net1] \
  -static_probability 0.2 -toggle_rate 10 -period 1000
```

To specify that the value of the net `net1` is logic 1 for 20% of the time, and that it transitions between logic values 0 and 1 an average of 10 times in 1000 time units. The time unit used for the toggle rate is the target library time unit. The `-period` option is optional and a default value of 1 is used when it is not specified.

State-dependent toggle rates can be annotated using the `-state_dep` option, as shown in this example:

```
set_switching_activity [get_pin ff1/Q] -toggle_rate 0.01 \
-state_dep "D"
set_switching_activity [get_pin ff1/Q] -toggle_rate 0.03 \
-state_dep "! D"
```

Which specifies that the pin `ff1/Q` toggles 0.01 times when the pin `D` is at logic 1, and 0.03 times when the pin `D` is at logic 0. The `-rise_ratio` option can also be used with state dependent toggle rates to specify the ratio of rise transitions to fall transitions for the specified state. For example:

```
set_switching_activity [get_pin xor1/Y] -toggle_rate 0.01 \
-state_dep "A" -rise_ratio 0.9
```

Specifies that the pin `xor1/Y` toggles 0.01 times when the cell is in state "A", and that 90% of these toggles are rise toggles.

Path-dependent toggle rates can be annotated using the `-path_dep` option as shown in the following example:

```
set_switching_activity [get_pin and1/Y] -toggle_rate 0.02 \
-path_dep "A"
set_switching_activity [get_pin and1/Y] -toggle_rate 0.00 \
-path_dep "B"
```

Specifies that the pin `and1/Y` toggles 0.02 times due to a toggle on the input pin `A`, but never toggles due to a toggle on `B`. Toggle rates that are both state and path dependent can be specified using the `-state_dep` and `-path_dep` options together.

State-dependent static probabilities can be annotated using the `-state_dep` option. The following:

```
set_switching_activity [get_cell and1] -static 0.1 \
-state_dep "A & B"
set_switching_activity [get_cell and1] -static 0.7 \
-state_dep "A & !B"
set_switching_activity [get_cell and1] -static 0.2 \
-state_dep "! A"
```

Specifies that the cell `and1` is at state "A & B" for 10% of the time, at state "A & !B" for 70% of the time, and at state "!A" for 20% of the time.

When you use the `set_switching_activity` command to annotate switching activity on all inputs, this will include the clock inputs as well. This results in overriding the switching activity on the clock inputs. To avoid overriding of switching activity on clock inputs, specify all inputs except the clock inputs. You can specify this as follows:

```
set_switching_activity [remove_from_collection [all_inputs] clk] \
-static_probability sp_value -toggle_rate tr_value -period period_value
```

More information about the `set_switching_activity` command syntax is:

```
set_switching_activity
    [-static_probability sp_value]
    [-toggle_rate tr_value]
    [-period period_value]
    [-clock clock_name]
    [-state_dep state_condition]
    [-path_dep path_sources]
    [-rise_ratio ratio_value]
    [-select select_types]
    [-hier]
    [-instances instances]
    [object_list]
    [-verbose]
```

For more information, see the man page.

Fully Annotating Versus Partially Annotating the Design

For the highest accuracy of power analysis, annotate all the elements in your design. To annotate all design elements, you must use gate-level simulation to monitor all the nodes of the design.

Using gate-level simulation, you can perform the following activities:

- Capture state- and path-dependent switching activity
- Capture switching activity that considers glitching (full-timing gate-level simulation only)

After layout, you can increase accuracy further by annotating wire loads. The improved accuracy results from the incorporation of more-accurate net capacitance values into power analysis. However, if the design layout is performed at the foundry, you might not have access to the post-layout information.

If you annotate some design elements, Power Compiler uses an internal zero-delay simulation to propagate switching activity through nonannotated nets in your design. Power Compiler uses internal simulation anytime it encounters nonannotated nets during power analysis.

Power Compiler always uses the most accurate switching activity available. During switching activity propagation, Power Compiler tracks which design elements are user-annotated with the `set_switching_activity` command and which are not. In calculating power, Power Compiler never overwrites user-annotated switching activity with propagated switching activity.

Power analysis and optimization require that you annotate at least the following:

- Primary inputs
- Outputs of synthesis-invariant elements such as black box cells
- Three-state devices
- Sequential elements
- Hierarchical ports

Note:

When performing power analysis on a partially annotated design, be sure to specify a clock before invoking the `report_power` command. The Power Compiler internal zero-delay simulation requires a real or virtual clock to properly compute and propagate switching activity through your design. Use the `create_clock` command in `dc_shell` to create a clock.

Analyzing the Switching Activity Annotation

The `report_saif` command can be used to display information about the annotated switching activity. The report generated by the `report_saif` command shows how much of the design objects is annotated with user-annotated switching activity, default switching activity, and propagated switching activity. The `report_saif` command considers clock-gating cells as synthesis invariant because they can be deleted or inserted during the optimization step.

```
dc_shell> report_saif
```

```
*****
Report : saif
Design : des
Version: 2004.12
Date:    August 26, 2009 8:08 am
*****
```

Object type	User Annotated (%)	Default Annotated (%)	Propagated Activity (%)	Total
Nets	251 (99.21%)	1 (0.40%)	1 (0.40%)	253
Ports	59 (98.33%)	1 (1.67%)	0 (0.00%)	60
Pins	251 (99.60%)	0 (0.00%)	1 (0.40%)	252

If the `-hier` or `-flat` option is used, the switching activity information is generated for all design objects in the design hierarchy starting from the current instance. If these options are missing, then only design objects in the hierarchical level of the current instance are considered.

If the `-rtl_saif` or `-type RTL` option is used, switching activity information about RTL invariant objects is printed. Otherwise switching activity information about all design nets, ports and pins are printed. You can use the `-rtl_saif` option after reading an RTL SAIF file.

The `-missing` option can be used to display the design objects that do not have user-annotated switching activity information.

Removing Switching Activity Annotation

Switching activity annotation can be removed from individual design objects using the `set_switching_activity` command. The following example shows the usage of this command:

```
dc_shell> set_switching_activity objects
```

Removes the simple and state- and path-dependent switching activity annotation from the specified objects.

Switching activity annotation can be removed from all the current design objects using the `reset_switching_activity` command. This command removes all the simple and state- and path-dependent switching activity information.

It is recommended that switching activity information from previous switching activity annotation is removed using the `reset_switching_activity` command before reading new SAIF files. For example, this illustrates a flow where an RTL SAIF file is read before a design is compiled with power constraints and then a more accurate gate-level SAIF file is used to generate power reports:

```
read_saif -input rtl.back.saif -instance tb_rtl/i
set_max_leakage_power 0 mW
set_max_dynamic_power 0 mW
compile_ultra
reset_switching_activity
read_saif -input gate.back.saif -instance tb_gate/i
report_power
```

Estimating Nonannotated Switching Activity

Power Compiler needs switching activity information about all design nets and state- and path-dependent information about all design cells and pins in order to calculate power. Switching activity that is not user annotated is estimated automatically before power is calculated. This is performed in three stages:

- Design nets whose switching activity can be calculated accurately or cannot be propagated are set to some default values. We say that such nets are default annotated.
- The user-annotated and default annotated switching activities are then used to derive the simple static probability and toggle rate information for the rest of the design nets.
- The simple switching activity information (user-annotated or estimated) is then used to derive the non-annotated state- and path-dependent switching activity.

Annotating Design Nets With Default Switching Activity Values

Design nets are annotated with default switching activity values when the switching activity can be accurately derived or when the switching activity cannot be estimated using the propagation mechanism described below. The first type of nets include nets driven by clocks since the switching activity information can be accurately derived from the clock waveform. For the second type of nets, it should be noted that the propagation mechanism uses the functionality of design cells to propagate the input switching activity to the cell outputs. Black box cells have unknown functionality, and therefore the switching activity of block-box outputs cannot be derived using the propagation mechanism. Outputs of the black box cells that are not user annotated is annotated with a default value.

The following lists all the different types of design nets that are annotated by default values:

- Nets driven by constants: A default toggle rate value of 0.0 is used. A static probability value of 0.0 is used for logic 0 constants, while a value of 1.0 is used for logic 1 constants.
- Nets driven by clocks: The default values for the toggle rate and static probability are derived from the clock waveform.
- Nets driving or driven by buffers: If the buffer input or output net is user or default annotated, then the nonannotated buffer output or input is default annotated with the switching activity values on the annotated input or output.
- Nets driving or driven by inverters: If the inverter input or output net is user or default annotated, then the nonannotated inverter output or input is default annotated with the same toggle rate value, and with the inverted static probability value. If the annotated static probability value is sp then the inverted static probability value is $1.0 - sp$.

- Flip-Flop outputs: If a flip-flop cell has both Q and QN output ports and only one of the outputs is annotated, then the other output is default annotated with the same toggle rate value and with the inverted static probability value.
- Primary inputs and outputs of black box cells: The switching activity of primary inputs and outputs of the black box cells cannot be propagated. Default switching activity depending on the value of the `power_default_static_probability` and `power_default_toggle_rate` variables is used. The default static probability value is the value of the `power_default_static_probability` variable. The default toggle rate value is the value of the `power_default_toggle_rate` multiplied by the related clock frequency. The related clock can be specified using the `-clock` option of the `set_switching_activity` command. If no related clock is specified on the net, the clock with the highest frequency is used. The default value of `power_default_static_probability` variable is 0.5 and the default value of `power_default_toggle_rate` variable is 0.1.

Propagating Switching Activity

The switching activity of design nets that are not user or default annotated are then derived using a propagation mechanism. This mechanism is basically a zero delay simulator. Random simulation vectors are generated for the user and default annotated nets depending on the annotated toggle rate and static probability values. The zero delay simulator uses the functionality of the design cells and the random vectors to obtain the switching activity on nonannotated cell outputs.

The number of simulation steps performed by this mechanism depends on the analysis effort option applied to the `report_power` command. User and default annotated switching activity values are never overwritten by values derived by the propagation mechanism.

However, if a design net is not annotated with both toggle rate and static probability values, then the switching activity on this net cannot be used by the propagation mechanism. For such nets, the nonannotated value is estimated by the propagation mechanism.

Deriving State- and Path-Dependent Switching Activity

If an RTL SAIF file or a gate-level SAIF file without state- and path-dependent switching information is used to annotate the design switching activity, Power Compiler needs to estimate the required state- and path-dependent switching activity information. After obtaining the simple switching activity (from user annotation, or by switching activity propagation), Power Compiler estimates the state-dependent static probability information for every cell, and the state- and path-dependent toggle rate information for every cell pin. This information is obtained from the switching activities of each cell input and output pins.

Although the state- and path-dependent estimation mechanism produces fairly accurate power calculations, for the most accurate power results, use gate-level SAIF files with state- and path-dependent information.

6

Performing Power Analysis

The information in this chapter describes the Power Compiler power analysis engine and how to perform power analysis.

This chapter contains the following sections:

- [Identifying Power and Accuracy](#)
- [Performing Gate-Level Power Analysis](#)
- [Analyzing Power With Partially Annotated Designs](#)
- [Power Correlation](#)
- [Design Exploration Using Power Compiler](#)
- [Examining Other dc_shell Commands for Power](#)
- [Using a Script File](#)
- [Power Reports](#)

Overview

After capturing switching activity, mapping your design to gates, and annotating your design, you can invoke power analysis by using the `report_power` command. This command analyzes the power of your design.

By changing the current design or by using command options, Power Compiler can create power reports for the following:

- Modules
- Individual nets
- Individual cells
- The total design

For a detailed explanation of the `report_power` command, see [“Performing Gate-Level Power Analysis” on page 6-5](#).

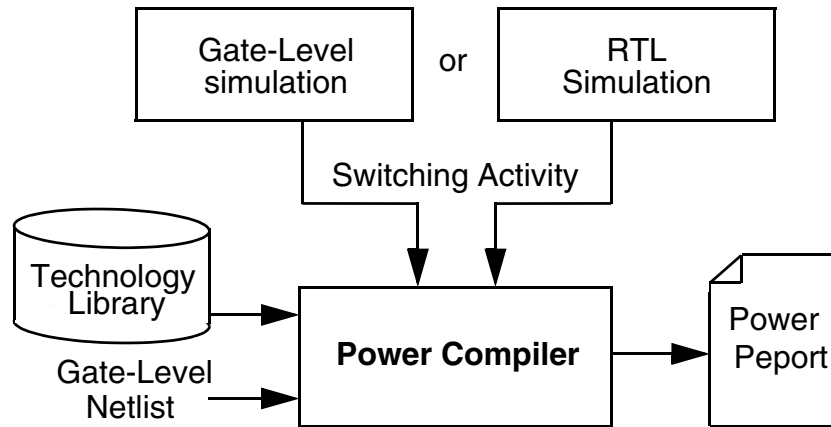
Identifying Power and Accuracy

Gate-level power analysis is always invoked with `report_power`. However, Power Compiler can use different methods to compute the power of your design. Power Compiler considers the type and amount of switching activity annotated on your design and chooses the most accurate method to compute your design’s power. Which method Power Compiler uses depends on whether you annotate some or all of the elements in your design.

To analyze your gate-level design, Power Compiler uses the following:

- switching activity
- technology library
- gate-level netlist

[Figure 6-1](#) shows the inputs to Power Compiler.

Figure 6-1 Information Inputs to Power Compiler

When you invoke power analysis, Power Compiler uses switching activity annotated on your design to compute power.

Your technology library should be characterized for power to show more accurate power results. If your technology library has pin capacitance and voltage and your technology library is not characterized for power, you can see power numbers using the `report_power` command. The numbers correspond to the switching power in the net. You are able to see power numbers because net switching power is a function of pin capacitance, voltage, and toggle frequency. It is recommended that you characterize your library for power.

Setting Accuracy Expectations

The following factors can affect the accuracy of power analysis:

- Switching activity annotation
- Delay model
- Correlation
- Clock tree buffers
- Complex cells

Switching Activity Annotation

Annotating switching activity relies on the ability to map the names of the synthesis invariant objects in the RTL source to the equivalent object names in the gate-level netlist. Mapping inconsistencies can cause the SAIF file to be incorrectly or incompletely annotated, which

can affect the power analysis results. In turn, the quality of these results affects the results of power optimizations that rely on the annotation, such as power-driven clock gating and operand isolation. For more information, see [“Annotating Switching Activity Using RTL SAIF Files” on page 5-2](#).

Delay Model

Power Compiler uses a zero-delay model for internal simulation and for propagation of switching activity during power analysis. This zero-delay model assumes that the signal propagates instantly through a gate with no elapsed time.

The zero-delay model has the advantage of enabling fast and relatively accurate estimation of power dissipation. The zero-delay model does not include the power dissipated due to glitching. If your power analysis must consider glitching, use power analysis after annotating switching activity from full-timing gate-level simulation. As mentioned previously, the internal simulation is used only for nodes that do not have user-annotated switching activity.

Correlation

While propagating switching activity through the design, Design Power makes certain statistical assumptions. However, the logic states of gates' inputs can have interdependencies that affect the accuracy of any statistical model.

Such interdependency of inputs is called correlation. Correlation affects the accuracy of Power Compiler propagation of toggle rates. Because accurate analysis depends on accurate toggle rates, correlation also affects the accuracy of power analysis.

Power Compiler considers correlation within combinational and sequential logic, resulting in more accurate analysis of switching activity for many types of designs. The types of circuits that exhibit high internal correlation are designs with reconvergent fanouts, multipliers, and parity trees. However, Power Compiler has no access to information about correlation external to the design. If correlation exists between the primary inputs of the design, Power Compiler does not recognize the correlation.

Power Compiler considers correlation only within certain memory and CPU thresholds, beyond which correlation is ignored. As the design size increases, Power Compiler reaches its memory limit and is not able to fully consider all internal correlation.

As an example of correlation, consider a 4-bit arithmetic logic unit (ALU) that performs five instructions. The data bus is 4-bits wide, and the instruction opcode lines are 3-bits wide. The assumption of uncorrelated inputs holds up well for the data bus lines inputs but fails for the opcode inputs if some instructions are used more often.

Clock Tree Buffers

The `set_cell_internal_power` command sets or removes the `power_value` attribute on or from specified pins. `power_value` is the value with which to set the `power_value` attribute, and represents the power consumption for a single toggle of the pin. If a cell has at least one such annotated pin, its internal power is calculated by summing the annotated power values times the pin toggle rates. If this command is issued without the `power_value` option, any existing `power_value` attributes are removed from the specified pins. If the `power_value` option is specified without `unit`, the power unit of the library is used. If the library does not have a defined unit, an error message is generated.

Use this command to override a cell's library power characterization in situations where that characterization does not apply; most commonly, when you manually replace an entire cloud of logic with a single cell and want the single cell's power consumption to represent that of the cloud of logic. For example, if you replace a clock tree by a single buffer cell, you can set the `power_value` attribute on the output pin of the buffer cell with the value of the power consumption for one clock toggle of the entire clock tree. Although the buffer cell might have been power-characterized in the library, its power consumption is now calculated using the value of the `power_value` attribute set by the `set_cell_internal_power` command. The syntax is

```
set_cell_internal_power pin_names  
[power_value ][unit]]
```

For more information, see the man page.

Complex Cells

If your design has black boxes, such as complex cells, RAM, ROM, or macrocells, annotate switching activity at the outputs of these elements.

Annotate the outputs of sequential elements. Power Compiler cannot initialize sequential elements in your design. Without annotation, Power Compiler cannot accurately propagate switching activity through sequential elements.

Performing Gate-Level Power Analysis

After annotating your design with switching activity, use the `report_power` command to analyze the power of your gate-level design.

From within the `dc_shell`, the `report_power` command checks out a Power Compiler license before analyzing the power of your design. If a license is not available, the command terminates with an error message. At the completion of the analysis, the Power Compiler license is released.

To keep the license at the completion of the `report_power` command, set the following:

```
power_keep_license_after_power_commands = "true"
```

This variable is valid only in `dc_shell`.

Note:

When performing power analysis on a partially annotated design, specify a clock before invoking the `report_power` command. The Power Compiler internal zero-delay simulation requires a real or virtual clock to properly compute switching activity. Use the `create_clock` command in `dc_shell` to create a clock.

Using the `report_power` Command

The `report_power` command calculates and reports power for a design. Power Compiler zero-delay simulation propagates switching activity for nets that are not user-annotated with switching activity. During the propagation, `report_power` uses the switching activity for startpoint nets (if available) when computing the switching activity for internal nets. The switching activity of any nets that are annotated with the `set_switching_activity` command is retained (it is not overwritten during the switching activity propagation).

If you annotate switching activity on all the elements of the design, Power Compiler does not propagate any switching activity through the design. Instead, power analysis uses the annotated gate-level switching activity.

Command options enable you to print with different sorting modes and with verbose and cumulative options. The default operation is to print a power summary for the instance's subdesign (in the context of the higher-level design).

Power analysis uses any net loads during the power calculation. For nets that do not have back-annotated capacitance, Power Compiler estimates the net load from the appropriate wire load model from the technology library. If you have annotated any cluster information about the design, using Synopsys Floorplan Manager, Power Compiler uses the improved capacitance estimates from the cluster's wire loads.

In the topographical mode the `report_power` command reports the correlated power of the design as a sum of estimated clock tree power and netlist power. For more details see ["Power Reports" on page 6-15](#).

Use the following syntax for the `report_power` command:

```
report_power [-net] [-cell]
              [-only cell_or_net_list]
              [-verbose] [-cumulative] [-flat]
              [-analysis_effort low | medium | high]
              [-include_input_nets]
              [-nworst number] [-sort_mode mode]
```



```
[-histogram ]
[-exclude_leq le_val | -exclude_geq ge_val]]
[-nosplit] [-hier] [-hier_level level_value]
[-scenario {scenario_name1 scenario_name2 ...}]
```

For more information, see the man page.

The default sort mode for `report_power -cell` is `cell_internal_power`. If the technology library does not have any internal power modeling for leaf cells, `report_power -cell -nworst 10`, for example, retrieves only the first ten cells (alphabetically). To change the sorting to something other than `cell_internal_power` sorting, use the `-sort_mode` option. The default sort mode for `report_power -net` is `net_switching_power`. If both the `-net` and `-cell` options are specified and a sort mode is explicitly specified, the selected sort mode is used for both the cell and net reports. Therefore, the selected sort mode must be one of the sort modes that applies to both options. If both the `-net` and `-cell` options are specified, by default, the sort mode for `report_power` is total dynamic power.

`-histogram [-exclude_leq le_val | -exclude_geq ge_val]`

This option prints a histogram-style report with the number of nets in each power range. Use the `-exclude_leq` and `-exclude_geq` options respectively to exclude data values less than *le_val* or greater than *ge_val*. This option is useful for printing the range and variation of power in the design and prints a histogram report only when used in conjunction with `-net` or `-cell` options.

`-nosplit`

Most of the design information is listed in fixed-width columns. If the information for a field exceeds its column width, the next field begins on a new line, starting in the correct column. This option prevents line splitting and facilitates scripts to extract information from the report output.

`-hier`

This option enables you to view internal, switching, and leakage power consumed in your design hierarchy on a block-by-block basis. The hierarchical levels of the design are indicated by indentations.

`-hier_level level_value`

Use this option only with the `-hier` option. This option enables you to limit the depth of the hierarchy tree displayed in the report. The *level_value* setting should be an integral number greater than or equal to 1. For example, to see the power results for all blocks up to 2 levels from the top, enter

```
report_power -hier -hier_level 2
```

`-scenario`

This option reports the power details for the specified list of scenarios for a multimode design. Inactive scenarios are not reported. When this option is not used, only the current scenario is reported.

Using the `report_power_calculation` Command

Power Compiler uses a complex mechanism to calculate dynamic and leakage power. The dynamic power consists of internal power on pins and switching power on nets. Both internal and leakage power could be state dependent.

Though the `report_power` command does provide a comprehensive report, it is often a mystery how the numbers relate to the power tables in the library.

The `report_power_calculation` command shows how the reported power numbers are derived from the various inputs such as library, simulation data, netlist, and parasitics etc. This command does not work on the libraries that have build-in security to protect the power table numbers. This does not apply for switching power. For more information, see the man page.

Analyzing Power With Partially Annotated Designs

If you invoke power analysis without annotating any switching activity, Power Compiler uses the following defaults for the primary inputs of your design:

- $P_1 = 0.5$ (the signal is in the 1 state 50 percent of the time)

P_1 is the probability that input P is at logic state 1. For definitions of static probability, P_1 , and toggle rate (TR), see [“Switching Activity That You Can Annotate” on page 5-2](#).

- $TR = 0.5 f_{clk}$ (the signal switches once every 2 clock cycles)

f_{clk} is the frequency of the input's related clock in the design, as defined by the `set_switching_activity` command. You can specify the related clock explicitly with its clock name or implicitly as “*”. In the latter case, Power Compiler infers a related clock automatically. If the input port does not have a related clock, Power Compiler uses the fastest clock in the design.

Using the defaults for static probability and toggle rate can be reasonable for data bus lines. However, the defaults might be unacceptable for some signals, such as a reset or a test-enable signal.

If you neglect to annotate toggle information about primary inputs, these inputs assume the default toggle value. If the input or logic connected to this input is heavily loaded, the results could be significantly different from what you expect.

To change the default values for switching activity and static probability, set the following variables to the values you want:

- `power_default_static_probability`

This variable sets the default value for static probability.

- `power_default_toggle_rate`

This variable sets the default value for toggle rate.

- `power_default_toggle_rate_type`

The default is `fastest_clock`, which causes Power Compiler to calculate the default toggle rate by multiplying the fastest clock's frequency with `power_default_toggle_rate`. Set this variable to `absolute` to determine the behavior when the design object does not have a specified related clock; Power Compiler simply uses the value of the `power_default_toggle_rate` variable.

The variables remain in effect throughout the `dc_shell` session in which you set them.

The following example sets the default static probability to 0.3:

```
set power_default_static_probability 0.3
```

The following example sets the default toggle rate to 0.4 of the toggle rate of the highest-frequency clock:

```
set power_default_toggle_rate 0.4
```

Power Correlation

Note:

This section pertains to Design Compiler topographical mode only.

Power correlation refers to the relationship between two power calculations: power after logic synthesis and power after place and route (P&R). Power after P&R is the final power, and you may want to know this number early in the process so you can take corrective action if the number exceeds your limits.

In `dc_shell`, the power reported after logic synthesis is often significantly different from the final power, and is, therefore, not a good predictor for final power. This differential is caused by three factors:

- Logic synthesis uses wire load models.
- High fanout nets are not synthesized.
- Clock trees do not exist in the design at the time of synthesis.

Performing logic synthesis within the Design Compiler topographical domain shell addresses the first two factors because this shell uses a virtual layout, not wire load models, and high fanout nets are synthesized automatically.

You specify to perform clock-tree estimation within `dc_shell-topo` to eliminate the differential caused by the third factor.

To improve correlation in cases with abnormal floor plans, you should use the physical constraints extracted from the floor plan.

Performing Power Correlation

Correlated power refers to design power added to estimated clock-tree power after logic synthesis in `dc_shell-topo`; this is also called estimated total power. Specifying any of the following power optimization commands correlates power:

- `compile_ultra -gate_clock`
- `set_max_dynamic_power`
- `set_max_leakage_power`
- `set_power_prediction`

The `set_power_prediction` command is only available in Design Compiler topographical mode and you must specify this command before the `compile_ultra` command. The syntax is:

```
set_power_prediction true | false  
    [-ct_references list_of_buffers_and_inverters]
```

Optionally, specify clock-tree references for clock-tree estimation to improve your correlation results.

Power Correlation Script

The following sample script correlates power after you have setup your design environment and applied synthesis constraints:

```
read_verilog  
set_power_prediction  
compile_ultra  
report_power  
write -f ddc -o design.ddc
```

In `dc_shell-topo`, the `report_power` command reports estimated total power, which includes the clock-tree contributions for internal, net-switching, and leakage power.

The `report_power` command does not report power if you don't use one of the power-estimation commands; Power Compiler issues the following error:

```
Error: Can't report power unless Power Prediction mode is on. (PWR-621)
```

Design Exploration Using Power Compiler

To use Power Compiler for design exploration, follow these steps to get quick results from gate-level power analysis:

1. Create a SAIF file.

This step requires RTL simulation. For information, see [Chapter 4, “Generating Switching Activity Information Format Files.”](#)

2. Compile the design to gates, using your choice of compile options.

3. Annotate switching activity on primary inputs and other synthesis-invariant elements of the gate-level design.

For information about using SAIF files from RTL simulation to annotate switching activity, see [Chapter 4, “Generating Switching Activity Information Format Files.”](#)

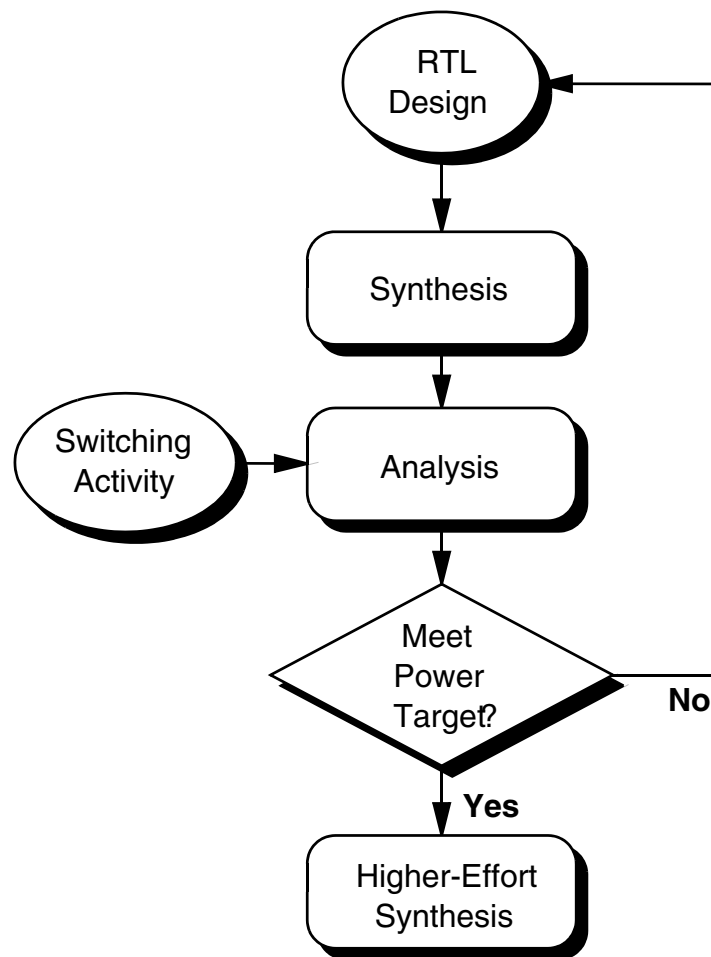
4. Use the `report_power` command to analyze your design's power.

Power Compiler uses an internal zero-delay simulation to propagate switching activity through nonannotated elements of the design.

5. Repeat steps 1 through 4 for other architectures and coding styles.

Quick gate-level power analysis enables you to see the results of changes in your RTL design.

[Figure 6-2](#) shows the steps that are followed in design exploration using Power Compiler.

Figure 6-2 Design Exploration Using Power Compiler

After you refine your RTL design within the iterative loop of design exploration, your design is ready for a higher-effort synthesis.

Examining Other `dc_shell` Commands for Power

Synopsys power products support the following `dc_shell` commands:

- `characterize`
- `report_lib`
- `write_script`

The `write_script` command creates a script file of a synthesis or analysis session. For more information about `write_script`, see the Design Compiler documentation and the man pages.

Characterizing a Design for Power

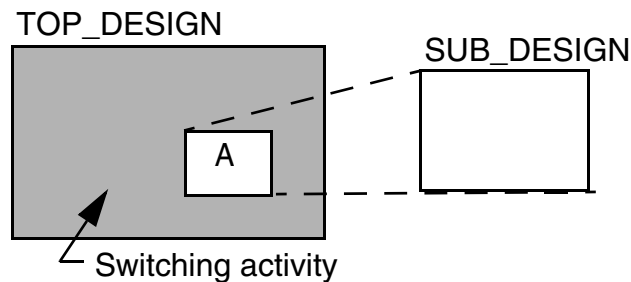
The `characterize` command has a particular option that is useful in power analysis and optimization: the `-power` option.

The `-power` option command characterizes annotated or propagated switching activity from the instance of a subdesign to the nets of the subdesign referenced by the instance. There must be a one-to-one correspondence between the nets in the instance and the nets in the referenced subdesign.

As shown in [Figure 6-3](#), consider a design hierarchy in which A is a design instance of SUB_DESIGN in TOP_DESIGN. Instance A references SUB_DESIGN. When you invoke power analysis on TOP_DESIGN, the switching activity propagates throughout any nets that are not already user-annotated.

```
dc_shell> report_power top_design
```

Figure 6-3 Switching Activity for TOP_DESIGN

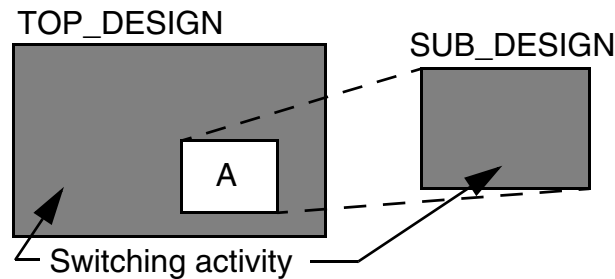


The switching activity can be propagated from primary inputs and synthesis-invariant elements. In this example, user-annotated on individual design elements using `set_switching_activity` commands, or both.

As shown in [Figure 6-4](#), if you set the current instance to A and characterize for power, `characterize` writes the switching activity of instance A onto SUB_DESIGN.

```
dc_shell> current_design TOP_DESIGN
dc_shell> characterize A -power
```

Figure 6-4 Switching Activity for SUB_DESIGN



After characterizing, you can report the power of SUB_DESIGN by using the newly characterized switching activity. If you have Power Compiler, you can compile the SUB_DESIGN by using the newly characterized switching activity.

The `-power` option of `characterize` relies on a one-to-one correspondence between the nets of the referenced SUB_DESIGN and its instance A. If you compile the subdesign before characterizing instance A or make any changes that alter the nets or names of nets, the one-to-one net correspondence is lost and `characterize` fails.

After compiling a subdesign and before reanalyzing or compiling TOP_DESIGN, be sure to relink the designs.

Before recompiling the subdesign, you might need to do some or all of the following steps:

- Relink the designs using `link`.
- Generate new switching activity for changed designs.
- Annotate or propagate new switching activity on designs.
- Characterize before reanalyzing or recompiling the subdesign.

For more information about the `characterize` command, see the Design Compiler documentation and the online man pages.

Reporting the Power Attributes of Library Cells

Use the `report_lib -power` command to report which library cells have power characterization and what type of characterization exists on each library cell. The `report_lib -power` command reports the following information for each cell:

- Leakage power attribute
- Internal power attribute

- Attribute for separate rise and fall power
- Attribute for average rise and fall power
- Toggling pin specified by the internal power table
- Any when conditions (for state-dependent power)
- The `related_pin` or `related_input` for path-dependent power

For more information about library commands, see the Library Compiler documentation or the man pages for individual commands.

Using a Script File

You can enter power analysis commands directly at the `dc_shell` prompt. However, many designers find it convenient to use a script file that contains commands for analysis or optimization.

The `dc_shell include` command executes a script file of commands.

Example

```
dc_shell> include script_file.scr
```

You can use `include` at the `dc_shell` prompt or from within another script file.

The `write_script` command can help you generate scripts. For specific information about `write_script`, see the Design Compiler documentation or man pages.

Power Reports

This section contains examples of reports generated with the `report_power` command and various combinations of report options.

The `report_power` command in topographical mode is enhanced to report the correlated power as a breakdown of estimated clock tree power and netlist power. If the tool cannot perform clock tree estimation, Power Compiler reports a warning that the clock tree estimation could not be performed.

Power Report Summary

[Example 6-1](#) shows a power report summary.

Example 6-1 Summary Report of the report_power Command

```
dc_shell> report_power -analysis_effort high -verbose
```

```
*****
Report : power
        -analysis_effort high
        -verbose
Design : DESIGN_1
Version: A-2007.12-SP2
Date   : Fri Feb 22 01:46:34 2008
*****

Library(s) Used:
    slow (File: slow.db)

Operating Conditions:
Wire Loading Model Mode: Inactive

Global Operating Voltage = 1.62
Power-specific unit information :
    Voltage Unit = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW      (derived from V,C,T units)
    Leakage Power Units = 1nW
```

Cell Internal Power Breakdown

```
-----
Combinational      =   3.0975 mW   (10%)
Sequential         =  22.3222 mW   (72%)
Other              =   0.0000 mW   (0%)
```

```
Combinational Count =   13470
Sequential Count    =    2382
Other Count         =         0
```

Information: Reporting correlated power. (PWR-620)

```
Cell Internal Power = 27.2572 mW   (76%)
Net Switching Power =  8.6208 mW   (24%)
-----
```

```
Total Dynamic Power = 35.8779 mW   (100%)
Cell Leakage Power   =  2.6586 uW
```

Power Breakdown

```
-----
```

	Cell Internal Power (mW)	Driven Net Switching Power (mW)	Tot Dynamic Power (mW) (% Cell/Tot)	Cell Leakage Power (pW)
Netlist Power	25.4197	5.5186	3.094e+01 (82%)	2.649e+03
Estimated Clock Tree Power	1.8375	3.1021	4.9396 (37%)	9.9143

```
-----
```

Net Power Report

[Example 6-2](#) shows a net power report sorted by `net_switching_power` and filtered to display only the five nets that have the highest switching power.

Example 6-2 Net Power Report, Sorting and Display Options

```
dc_shell> report_power -net -flat -sort_mode
net_switching_power -nworst 5
```

```
*****
Report:  power
        -net
        -nworst 5
        -flat
        -sort_mode net_switching_power
Design:  DESIGN_1
Version: A-2007.12-SP2
Date   : Fri Feb 22 01:50:50 2008
*****
Library(s) Used:

        power_lib.db (File: /remote/libraries/power_lib.db)

Operating Conditions: slow   Library: slow
Wire Load Model Mode: Inactive.
```

```
Global Operating Voltage = 1.62
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 = 1nW
```

Attributes

```
-----
a - Switching activity information annotated on net
d - Default switching activity information on net
```

Net	Total Net Load	Static Prob.	Toggle Rate	Switching Power	Attrs
U_TAP_DBG_U_DBG_net5051	0.463	0.374	0.1968	0.1195	
U_CORE/U_CONTROL_U_A7S_pencadd_net5225	0.248	0.374	0.1968	0.0641	
U_CORE/U_CONTROL_U_A7S_dataio_net5298	0.247	0.374	0.1968	0.0637	
U_CORE/U_MUL8_net5450	0.232	0.374	0.1968	0.0599	
U_CORE/U_AREG_net5593	0.194	0.374	0.1968	0.0501	
Total (5 nets)				357.2614 uW	

Cell Power Report

[Example 6-3](#) displays a cell power report containing the cumulative cell power report. The cells are sorted by cumulative fanout power values, and only the top five are reported.

Example 6-3 Cell Power Report Containing Cumulative Cell Power

```
dc_shell> report_power -cell -analysis_effort low
-sort_mode cell_internal_power
```

```
*****
Report : power
        -cell
        -analysis_effort low
        -sort_mode cell_internal_power
Design : DESIGN_3
Version: B-2008.09
Date   : Fri Aug 08 01:51:28 2008
*****
```

Library(s) Used:

slow (File: slow.db)

Operating Conditions: slow Library: slow
Wire Load Model Mode: Inactive.

Global Operating Voltage = 1.62
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 = 1nW

Information: Reporting correlated power. (PWR-620)

Attributes

h - Hierarchical cell

Cell	Cell Internal Power	Driven Net Switching Power	Tot Dynamic Power (% Cell/Tot)	Cell Leakage Power	Attrs
CLOCK_TREE_EST	1.8375	3.1021	4.940 (37%)	9.9144	
U_CORE	21.7118	N/A	N/A (N/A)	2226.6487	h
U_TAP_DBG_U_DBG_clk_gate_int_en_d_reg	0.0123	N/A	N/A (N/A)	1.4392	h
0.0112 6.968e-04 1.19e-02 (94%)		0.1458			
U_TAP_DBG_U_SCAN1_breakpt_in_d_reg	0.0106	2.472e-04	1.09e-02 (98%)	0.1458	

U_TAP_DBG_U_ID_REG_clk_gate_shift_reg

...

Totals (2474 cells) 27.368mW N/A N/A (N/A) 2.658uW

Hierarchical Power Reports

These examples show the results of the `report_power` command with the hierarchical options. [Example 6-4](#) shows the results of the `report_power` command using the `-hier` option. This option shows the internal, switching, and leakage power consumed in your design hierarchy on a block-by-block basis.

Example 6-4 Hierarchical `report_power` with `-hier` Option

```
dc_shell> report_power -hier
```

```
*****
Report : power
       -hier
       -analysis_effort low
Design : DESIGN_4
Version: A-2007.12-SP2
Date   : Fri Feb 22 01:51:42 2008
*****
```

Library(s) Used:

slow (File: slow.db)

Operating Conditions: slow Library: slow
Wire Load Model Mode: Inactive.

Global Operating Voltage = 1.62
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 = 1nW

Information: Reporting correlated power. (PWR-620)

Hierarchy	Switch Power	Int Power	Leak Power	Total Power	%
A7S_top	8.683	27.368	2.66e+03	36.054	100.0
CLOCK_TREE_EST	3.102	1.837	9.914	4.940	13.7
U_CORE (A7S_core)	4.318	21.712	2.23e+03	26.032	72.2

[Example 6-5](#) shows the results of the `report_power` command using the `-hier` and `-hier_level` options. The `-hier` option shows the internal, switching, and leakage power consumed in your design hierarchy on a block-by-block basis. The `-hier_level` option limits the depth of the hierarchy level displayed in the report.

Example 6-5 Hierarchical report_power With -hier and -hier_level Options

```
dc_shell> report_power -hier -hier_level 1
```

```
*****
Report : power
        -hier
        -analysis_effort low
Design  : A7S_top
Version: A-2007.12-SP2
Date    : Fri Feb 22 01:51:42 2008
*****

Library(s) Used:

    slow (File: slow.db)

Operating Conditions: slow   Library: slow
Wire Load Model Mode: Inactive.

Global Operating Voltage = 1.62
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 = 1nW
```

Information: Reporting correlated power. (PWR-620)

Hierarchy	Switch Power	Int Power	Leak Power	Total Power	%
A7S_top	8.683	27.368	2.66e+03	36.054	100.0
CLOCK_TREE_EST	3.102	1.837	9.914	4.940	13.7
U_CORE (A7S_core)	4.318	21.712	2.23e+03	26.032	72.2

Power Report For Interface Logic Model

The `report_power` command can report the total power information for interface logic models. The tool reports the power information of the ILM by default unless you set the `ilm_enable_power_calculation` variable to false before creating the ILM. The default value of this variable is true. For more details see the Using Interface Logic Models Chapter in the Design Compiler User Guide.

No specific command line options are required to report the power information for Interface Logic Model.

The reporting for ILMs can be done for both multivoltage and non-multivoltage designs and also for hierarchical flows. The following example shows a sample power report for ILM.

Example 6-6 report_power for Interface Logic Model

```
dc_shell> report_power
```

```
*****
```

```
Report : power
```

```
      -hier
```

```
Design : top
```

```
Version: A-2008.09
```

```
Date   : Fri Aug 22 01:51:42 2008
```

```
*****
```

```
Library(s) Used:
```

```
      slow (File: slow.db)
```

```
Operating Conditions: slow   Library: slow
```

```
Wire Load Model Mode: Inactive.
```

```
Global Operating Voltage = 1.62
```

```
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 = 1nW
```

```
Information: Reporting correlated power. (PWR-620)
```

```
  Cell Internal Power = 52.6285 mW(90%)
```

```
  Net Switching Power = 5.9982 mW(10%)
```

```
-----
```

```
Total Dynamic Power   = 58.6267 mW(100%)
```

```
Cell Leakage Power    = 706.2805 uW
```

```
Power Breakdown
```

```
-----
```

Cell	Cell Internal Power (mW)	Driven Net Switching Power (mW)	Tot Dynamic Power (mW) (% Cell/Tot)	Cell Leakage Power (pW)
Netlist Power	50.1131	4.9944	5.511e+01 (91%)	6.978e+08
Estimated Clock Tree Power	2.5155	1.0037	3.519e+00 (71%)	8.463e+06

```
-----
```


7

Clock Gating

Power optimization at high levels of abstraction has a significant impact on reduction of power in the final gate-level design. Clock gating is an important high-level technique for reducing the power consumption of a design.

This chapter includes the following sections:

- [Introduction to Clock Gating](#)
- [Using Clock-Gating Conditions](#)
- [Inserting Clock Gates](#)
- [Specifying Clock-Gate Latency](#)
- [Calculating the Clock Tree Delay From Clock-Gating Cell to Registers](#)
- [Specifying Setup and Hold](#)
- [Choosing Gating Logic](#)
- [Selecting Clock-Gating Style](#)
- [Modifying the Clock-Gating Structure](#)
- [Integrated Clock-Gating Cells](#)
- [Propagating Clock Constraints](#)
- [Ensuring Accuracy When Using Ideal Clocks](#)

- [Sample Clock-Gating Script](#)
- [Clock-Gating Naming Conventions](#)
- [Keeping Clock-Gating Information in a Structural Netlist](#)
- [Replacing Clock-Gating Cells](#)
- [Optimizing Clock-Gate Insertion](#)
- [Performing Clock-Gating on DesignWare Components](#)
- [Reporting Command For Clock Gates and Clock Tree Power](#)

Introduction to Clock Gating

Clock gating applies to synchronous load-enable registers, which are groups of flip-flops that share the same clock and synchronous control signals and that are inferred from the same HDL variable. Synchronous control signals include synchronous load-enable, synchronous set, synchronous reset, and synchronous toggle.

The registers are implemented by Design Compiler by use of feedback loops. However, these registers maintain the same logic value through multiple cycles and unnecessarily use power. Clock gating saves power by eliminating the unnecessary activity associated with reloading register banks.

Designs that benefit most from clock gating are those with low-throughput datapaths. Designs that benefit less from RTL clock gating include designs with finite state machines or designs with throughput-of-one datapaths.

Power Compiler allows you to perform clock gating with the following techniques:

- RTL-based clock gate insertion on unmapped registers. Clock gating occurs when the register bank size meets certain minimum width constraints.
- Gate-level clock gate insertion on both unmapped and previously mapped registers. In this case, clock gating is also applied to objects such as IP cores that are already mapped.
- Power-driven gate-level clock gate insertion, which allows for further power optimizations because all aspects of power savings, such as switching activity and the flip-flop types to which the registers are mapped, are considered.

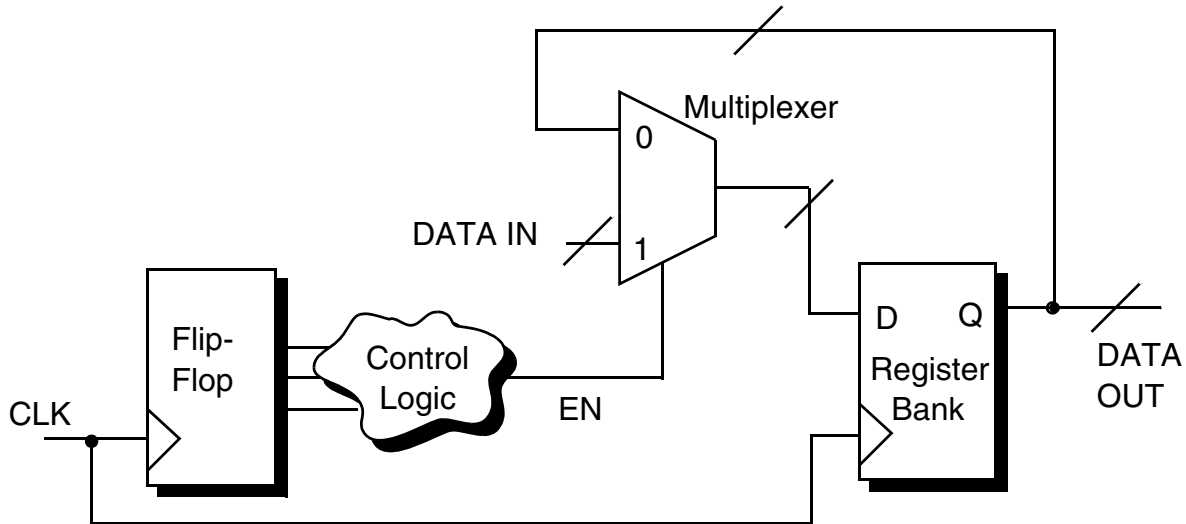
You can choose the type of clock-gating circuit inserted. Following are some of the choices:

- Choose an integrated or nonintegrated cell with latch-based clock gating
- Choose an integrated or nonintegrated cell with latch-free clock gating
- Insert logic to increase testability
- Specify a minimum number of bits below which clock gating is not inserted
- Explicitly include signals in clock gating
- Explicitly exclude signals from clock gating
- Specify a maximum number for the fanouts of each clock-gating element
- Move a clock-gated register to another clock-gating cell
- Resize the clock-gating element

Without clock gating, Design Compiler implements register banks by using a feedback loop and a multiplexer. When such registers maintain the same value through multiple cycles, they use power unnecessarily.

Figure 7-1 shows a simple register bank implementation using a multiplexer and a feedback loop.

Figure 7-1 Synchronous Load-Enable Register With Multiplexer



When the synchronous load enable signal (EN) is at logic state 0, the register bank is disabled. In this state, the circuit uses the multiplexer to feed the Q output of each storage element in the register bank back to the D input. When the EN signal is at logic state 1, the register is enabled, enabling new values to load at the D input.

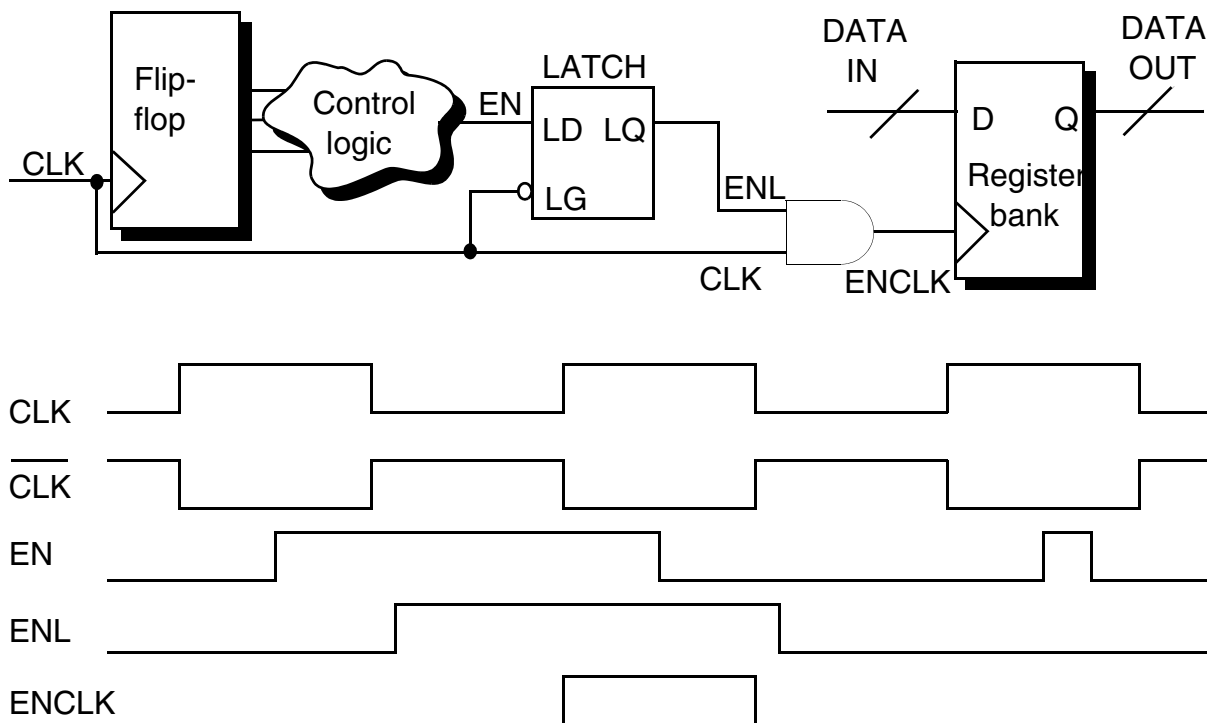
Such feedback loops can unnecessarily use power. For example, if the same value is reloaded in the register throughout multiple clock cycles (EN equals 0), the register bank and its clock net consume power while values in the register bank do not change. The multiplexer also consumes power.

Clock gating eliminates the feedback net and multiplexer shown in **Figure 7-1** by inserting a 2-input gate in the clock net of the register. Clock gating can insert inverters or buffers to satisfy timing or clock waveform polarity requirements.

The 2-input clock gate selectively prevents clock edges, thus preventing the gated-clock signal from clocking the gated register.

Figure 7-2 shows a latch-based clock-gating style using a 2-input AND gate; however, depending on the type of register and the gating style, gating can use NAND, OR, and NOR gates instead.

Figure 7-2 Latch-Based Clock Gating



At the bottom of [Figure 7-2](#), waveforms of the signals are shown with respect to the clock signal, CLK.

The clock input to the register bank, ENCLK, is gated on or off by the AND gate. ENL is the enabling signal that controls the gating; it derives from the EN signal on the multiplexer shown in [Figure 7-1](#). The register bank is triggered by the rising edge of the ENCLK signal.

The latch prevents glitches on the EN signal from propagating to the register's clock pin. When the CLK input of the 2-input AND gate is at logic state 1, any glitching of the EN signal could, without the latch, propagate and corrupt the register clock signal. The latch eliminates this possibility because it blocks signal changes when the clock is at logic state 1.

In latch-based clock gating, the AND gate blocks unnecessary clock pulses by maintaining the clock signal's value after the trailing edge. For example, for flip-flops inferred by HDL constructs of rising-edge clocks, the clock gate forces the gated clock to 0 after the falling edge of the clock.

By controlling the clock signal for the register bank, you can eliminate the need for reloading the same value in the register through multiple clock cycles. Clock gating inserts clock-gating circuitry into the register bank's clock network, creating the control to eliminate unnecessary register activity.

Clock gating reduces the clock network power dissipation, relaxes the datapath timing, and reduces routing congestion by eliminating feedback multiplexer loops. For designs that have large multi-bit registers, clock gating can save power and reduce the number of gates in the design. However, for smaller register banks, the overhead of adding logic to the clock tree might not compare favorably to the power saved by eliminating a few feedback nets and multiplexers.

Using Clock-Gating Conditions

Before gating the clock signal of a register, Power Compiler checks if certain clock-gating conditions are satisfied. Power Compiler inserts a clock gate only if all clock-gating conditions are satisfied.

Registers in your design qualify for clock gating when the following conditions are met:

- The circuit demonstrates synchronous load-enable functionality.
- The circuit satisfies the setup condition.
- The register bank or group of register banks satisfies the minimum number of bits you specify with the `set_clock_gating_style -minimum_bitwidth` command. The default value used for the minimum bitwidth is 3.

After clock gating is complete, the status of clock-gating conditions for gated and ungated register banks appears in the clock-gating report. For information about the clock-gating report, see [“Reporting Command For Clock Gates and Clock Tree Power” on page 7-72](#).

Clock-Gating Conditions

The register must satisfy all three of the following conditions before Power Compiler gates the clock signal of the registers:

- Enable condition

This condition checks if the register bank's synchronous load-enable signal is constant logic 1, reducible to logic 1, or logic 0. In these cases, the condition is false and the circuit is not gated. If the synchronous load-enable signal is not constant logic 1 or 0, the condition is true and clock gating goes on to check the setup condition. The enable condition is the first condition clock gating checks.

- Setup condition

This setup condition applies to latch-free clock gating only. It checks that the enable signal comes from a register that is clocked by the same clock as the register being gated. Clock gating checks this condition only if the register satisfies the enable condition.

- Width condition

The width condition is the minimum number of bits for gating registers or groups of registers with equivalent enable signals. The default value is 3. You can set the width condition by using the `-minimum_bitwidth` option of the `set_clock_gating_style` command. Clock gating checks this condition only if the register satisfies the enable condition and the setup condition.

Enable Condition

The enable condition of a register or clock gate is a combinational function of nets in the design. The enable condition of a register represents the states for which a clock signal must be passed to the register. The enable condition of a clock gate corresponds to the states for which a clock is passed to the registers in the fanout of the clock gate. Power Compiler utilizes the enable condition of the registers for clock-gate insertion.

Enable conditions are represented by boolean expressions for nets. For example:

```
module TEST (en1, en2, en3, in, clk, dataout);
    input en1, en2, en3, clk;
    input [5:0] in;
    output [5.0] dataout;
    reg [5.0] dataout;

    wire enable;

    assign enable = (en1 | en3) & en2;

    always @( posedge clk ) begin
        if( enable )
            dataout <= in;
        else
            dataout <= dataout;
    end

endmodule
```

In this example, the enable condition for the register bank `dataout_reg*` can be expressed as `en1 en2 + en3 en2`.

Enable conditions can be hard to identify in the RTL netlist. Set the `power_cg_print_enable_conditions` variable to `true` to report the enable conditions. Control the number of boolean expressions included in the report with the `power_cg_print_enable_conditions_max_terms` variable. The default is 10.

Setup Condition

If the enable condition is satisfied, Power Compiler requires that the enable signal of the register bank enable be synchronous with its clock. This is the setup condition.

For latch-based or integrated clock gating, Power Compiler can insert clock gating irrespective of the enable signal's and the clock's clock domains. If the enable signal and the register bank reside in different clock domains, you must ensure that the two clock domains are synchronous and that the setup and hold times for the clock-gating cell meet the timing requirements.

For latch-free clock gating, if any of the following characteristics exist, the setup condition is false and the register bank is not gated:

- If the register bank and its controlling logic (including flip-flops) belong to different clock domains, the setup condition is false.
- If the register bank and its controlling logic (including flip-flops) are driven by different edges of the same clock signals, the setup condition is false.
- If the controlling logic is driven by a combination path from the input port, the setup condition is false, unless:
 - For primary input ports, you specified a clock with the `set_input_delay` command.
 - You specified `power_cg_derive_related_clock true`, which enables clock propagation of the related clocks from parent hierarchies for inputs on subdesigns. The default is `false`.

These two special cases specify that an input port is synchronous with a given clock; therefore, the setup condition is true.

Specify `power_cg_ignore_setup_condition true` to cause Power Compiler to ignore the setup condition for latch-free clock gating. Use this variable with extreme caution.

Overriding Clock-Gating Conditions

Use the `set_clock_gating_registers` command to explicitly include or exclude an HDL signal in clock gating, thus overriding the clock-gating conditions.

The syntax is

```
set_clock_gating_registers
  [-include_instances instance_list]
  [-exclude_instances instance_list]
  [-undo register_list]
```

For example, suppose you have two clocks, `clk1` and `clk2`, and you only want to perform clock gating on `clk2`. Specify the `set_clock_gating_registers` command as follows:

```
set_clock_gating_registers -exclude [all_reg]
set_clock_gating_registers -include [all_reg -clock clk2]
```


Use this command with the `compile_ultra -gate_clock` and the `insert_clock_gating` command.

For additional information about these commands, see the respective man pages.

Inserting Clock Gates

Power Compiler inserts clock-gating cells to your design if you compile your design using the `-gate_clock` option of the `compile` or `compile_ultra` command. You can also insert clock gates to your design using the `insert_clock_gating` command. The following sections discuss in detail these two ways of clock-gate insertion.

Using the `compile_ultra -gate_clock` Command

During the compilation process, Power Compiler can insert clock-gates to your design if you use the `-gate_clock` option of the `compile` or `compile_ultra` commands. With the `-gate_clock` option, `compile` or `compile_ultra` commands can perform clock-gate insertion on the gate-level netlist and the RTL netlist as well as GTECH netlist. By default, when you use the `-gate_clock` option, the tool inserts clock gates only in the same level of hierarchy as the registers gated by the clock gate. For the tool to perform clock gating across the design hierarchy, set the `compile_clock_gating_through_hierarchy` variable to `true`. For more details on hierarchical clock gating see [“Hierarchical Clock Gating” on page 7-68](#).

The `compile_ultra -gate_clock` command can also perform clock gating on DesignWare components. For more details, see [“Performing Clock-Gating on DesignWare Components” on page 7-72](#).

Using the `insert_clock_gating` Command

The `insert_clock_gating` command can be used to perform clock-gating on the GTECH netlist. You cannot use this command to perform clock gating on gate-level netlist. To perform clock gating on a gate-level netlist use the `compile_ultra -gate_clock` command. This command identifies clock-gating opportunities by combining different register banks that share common enable signal.

The `insert_clock_gating` command performs clock gating on all the subdesigns in the design hierarchy by processing each subdesign independently. Use the `-no_hier` option to limit the clock-gate insertion to the top level of the design hierarchy. Use the `-global` option to perform hierarchical clock gating, that is, to insert clock gates on all levels of design hierarchy, considering the design as a whole and not considering each subdesign

independently, . For more details on hierarchical clock gating see [“Hierarchical Clock Gating” on page 7-68](#). For more details of the `insert_clock_gating` command see the command man page.

Clock-Gate Insertion in Multivoltage Designs

In a multivoltage design, the different hierarchies of the design can have different operating condition definition and use different target library subsets. So, while inserting clock-gating cells in a multivoltage design, Power Compiler chooses the appropriate library cells based on the specified clock gating style as well as the operating conditions that match the operating conditions of the hierarchical cell of the design. If you do not specify a clock gating style, the tool chooses a suitable clock gating style. If the tool does not find a library cell that suites both, the clock gating style and the operating condition, a clock gating cell is not inserted and a warning message is issued. For more details on clock gating style see [“Selecting Clock-Gating Style” on page 7-33](#).

Clock Gating Flows

The various clock-gating flows supported by the tool is discussed in detail in the following sections.

Inserting Clock Gates in the RTL Design

To insert clock gating logic in your RTL design and to synthesize the design with the clock gating logic, follow these steps:

1. Read the RTL design.
2. Use the `compile_ultra -gate_clock` command to compile your design.

During the compilation process clock gate is inserted on the registers qualified for clock-gating. By default, during the clock-gate insertion the `compile_ultra` command uses the default values of the `set_clock_gating_style` command and also honors the setup, hold, and other constraints specified in the technology libraries. To override the setup and hold values specified in the technology library, use the `set_clock_gating_style` command before compiling your design.

You can also use the `insert_clock_gating` command to insert the clock-gating cells.

Both, `compile_ultra` and `insert_clock_gating` commands use the default values of the clock gating style during the clock gate insertion. The default values of the `set_clock_gating_style` command is suitable for most designs. For more details on the default clock-gating style, see [“Default Clock-Gating Style” on page 7-42](#)

3. If you are using testability in your design, use the `insert_dft` command to connect the `scan_enable` and the `test_mode` ports or pins of the integrated clock-gating cells.
4. Use the `report_clock_gating` command to report the registers and the clock gating cells in the design. Use the `report_power` command to get details of the dynamic power utilized by your design after the clock gate insertion.

In the following example, clock gating is implemented in the design during the compilation process. The default values of the `set_clock_gating_style` command are used during the clock-gate insertion. The `-scan` option of the `compile_ultra` command enables the examination of your design for scan insertion for mission mode constraints.

```
dc_shell> read_verilog design.v
dc_shell> create_clock -period 10 -name CLK
dc_shell> compile_ultra -gate_clock -scan
dc_shell> insert_dft
dc_shell> report_clock_gating
dc_shell> report_power
```

Inserting Clock Gates in Gate-Level Design

To insert clock gating logic in your gate-level netlist and to re-synthesize the design with the clock gating logic follow these steps:

1. Read the gate-level netlist.
2. Use the `compile_ultra -gate_clock` command to compile your design.

During the compilation process, clock-gating cells are inserted on the registers qualified for clock-gating. During this process by default, the `compile_ultra` command

- Reads the setup and hold constraints that are specified in the technology libraries
- Propagates these constraints up the hierarchy.

To override the setup and hold values specified in the technology library, use the `set_clock_gating_style` before compiling your design. Using the `compile_ultra -gate_clock` command you can perform clock-gate insertion on designware elements as well. For more details about clock-gate insertion on designware components see, [“Performing Clock-Gating on DesignWare Components” on page 7-72](#).

The `compile_ultra -gate_clock` command uses the default values of the clock gating style during the clock-gate insertion. The default values of the `set_clock_gating_style` command are suitable for most designs. For more details on the default clock-gating style, see [“Default Clock-Gating Style” on page 7-42](#)

3. If you are using testability in your design, use the `insert_dft` command to connect the `scan_enable` and `test_mode` ports or pins of the integrated clock-gating cells.

4. Use the `report_clock_gating` command to report the registers and the clock gating cells in the design. Use the `report_power` command to get details of the dynamic power utilized by your design after the clock gate insertion.

In the following example, clock gating is implemented in the design during the compilation process. The default values of the `set_clock_gating_style` command are used during the clock-gate insertion.

```
dc_shell> read_ddc design.ddc
dc_shell> compile_ultra -inc -gate_clock -scan
dc_shell> insert_dft
dc_shell> report_clock_gating
dc_shell> report_power
```

Power-Driven Clock Gating

You can perform power-driven clock gating by setting the `power_driven_clock_gating` variable to `true` before synthesizing your design. With this variable set, while compiling the design using the `compile_ultra -gate_clock` command, Power Compiler performs the following tasks:

- Examines all register banks that can potentially be clock-gated, calculates their power with and without clock gates, and retains the clock gates that provide lower power costs.
- Uses default or user-annotated switching activity information to help compute the power costs for clock gates inserted at the register banks.

If you do not specify the switching activity, the tool uses the default switching activity. You specify the switching activity either by SAIF or by using the `set_switching_activity` command. If you specify the switching activity, the nonannotated nodes have the propagated switching activity. For information about SAIF, see [“Annotating Switching Activity Using RTL SAIF Files” on page 5-2](#).

Propagation of the default switching activity assumes 50% toggle rate with respect to the clock. For more information about default propagation, see [“Annotating Design Nets With Default Switching Activity Values” on page 5-12](#).

- Performs clock gating with the necessary mapping optimizations.
- Optimizes new and existing clock-gating logic. Optimizations can be any of the following: insertion, removal, changing fanout, and combining redundant clock gates.
- Performs DesignWare clock gating.
- In Design Compiler topographical mode, automatically uses power correlation. For more information, see [“Power Correlation” on page 6-9](#).

Note:

Gate-level and power-driven clock gating work with the `compile_ultra -inc` option as well, but not if Design Compiler is in topographical mode.

Power-driven clock gating only considers the minimum bit-width if you specify it explicitly with the `set_clock_gating_style -min_bitwidth` command. Doing so, however, can interfere with the algorithm that evaluates the actual power savings of the clock gates. This algorithm is more accurate than relying on the minimum bitwidth and typically leads to better dynamic power results. The following warning message appears if you specify `set_clock_gating_style` with the `-minimum_bitwidth` option:

```
Warning: A minimum bit-width constraint has been set;
power-driven clock gating may yield inferior results.
(PWR-650)
```

The following script performs power-driven clock-gate insertion in Design Compiler:

```
#optional setting
set_clock_gating_style -positive integrated
read_verilog {register_bank.v subdesign.v top.v}
current_design top
link
create_clock -p 5 clk -name CLK
compile_ultra -gate_clock
insert_dft
report_clock_gating
report_power
```

[Table 7-1](#) compares the various clock-gating techniques:

Table 7-1 Clock-Gating Technique Comparison

	RTL	Gate-Level	Power-Driven
Input Netlist	RTL	Gate-level	Gate-level
Honor <code>set_clock_gating_style</code> command	Yes	Yes	Yes. Using the <code>-min_bitwidth</code> option is not recommended
Cost factor considered during clock-gate insertion	Design topology	Design topology	Dynamic power and switching activity
Performing Clock gate insertion	Use <code>compile_ultra -gate_clock</code> or <code>insert_clock_gating</code> commands	<code>compile_ultra -gate_clock</code> or <code>compile_ultra -inc -gate_clock</code>	<code>set power_driven_clock -gating true</code> followed by <code>compile -gate_clock</code>

Table 7-1 Clock-Gating Technique Comparison (Continued)

	RTL	Gate-Level	Power-Driven
Tasks performed	Clock-gate insertion	Clock-gate insertion	Clock-gate insertion, optimization, and removal
Additional tasks performed in Design Compiler Topographical Technology	None	Power correlation	Power correlation

Specifying Clock-Gate Latency

During synthesis, Design Compiler assumes that the clocks are ideal. An ideal clock incurs no delay through the clock network. This assumption is made because real clock-network delays are not known until after clock tree synthesis. In reality clocks are not ideal and there is a non-zero delay through the clock network. For designs with clock gating, the clock-network delay at the registers is different from the clock-network delay at the clock-gating cell. This difference in the clock-network delay at the registers and at the clock-gating cell results in stricter constraints for the setup condition at the enable input of the clock-gating cell.

For Design Compiler to account for the clock network delays during the timing calculation, specify the clock network latency using either the `set_clock_gate_latency` or the `set_clock_latency` command. The `set_clock_gate_latency` command can be used for both, gate-level and RTL designs. The `set_clock_latency` command can be used only on RTL netlist. More details of these two commands are described in the following sections.

The `set_clock_gate_latency` Command

When you use the `compile_ultra -gate_clock` command, clock gates are inserted during the compilation process. To specify the clock network latency, even before the clock-gating cells are inserted by the tool, use the `set_clock_gate_latency` command. This command lets you specify the clock network latency for the clock-gating cells as a function of the clock domain, clock gating stage, and the fanout of the clock-gating cell. The latency that you specify is annotated on the clock gating cells when they are inserted by the `compile_ultra -gate_clock` command. You can manually annotate the latency values on the clock-gating cells using the `apply_clock_gate_latency` command. For more details, see [“Applying Clock-Gate Latency” on page 7-17](#).

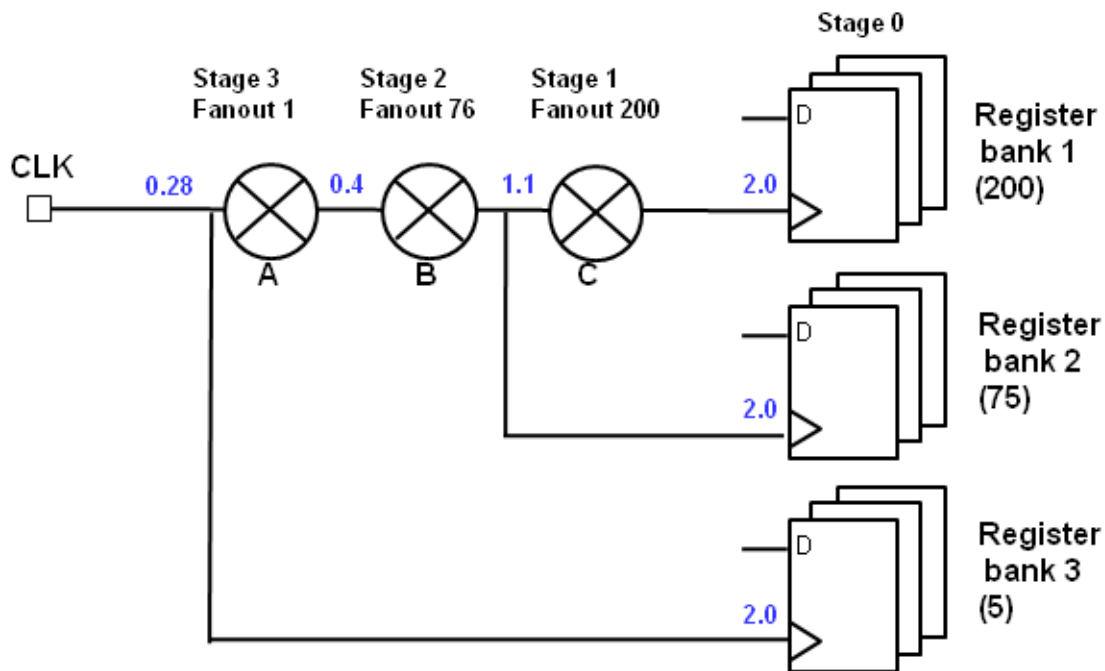
The following example in [Figure 7-3](#) shows the definitions for the clock-gate stages and the fanouts.

The clock gating cell C drives 200 registers. So the fanout of the cell C is 200. Because C drives registers, and not other clock gating cells, the clock gating stage for the cell C is 1.

The clock gating cell B drives a set of 75 registers and a clock gating cell C. So the fanout of the clock-gating cells B is 76. The clock gating stage for the cell B is 2; clock gating stage of C + 1.

Similarly, the clock gating stage of cell A is 3 and the fanout is 1. The clock gating stage of all the registers is stage 0.

Figure 7-3 Clock-Gating Stages and Fanouts



The following example script shows how to specify the latency values for the various clock gate stages and fanouts using the `set_clock_gate_latency` command for the design shown in [Figure 7-3](#).

```
set_clock_gate_latency -clock CLK -stage 0 \
  -fanout_latency {1-inf 2.0}
set_clock_gate_latency -clock CLK -stage 1 \
  -fanout_latency {1-30 2.1, 31-100 1.7, 101-inf 1.1}
set_clock_gate_latency -clock CLK -stage 2 \
```

```
-fanout_latency {1-5 0.9, 6-20 0.5, 21-100 0.4, 101-inf 0.3}
set_clock_gate_latency -clock CLK -stage 3 \
-fanout_latency {1-10 0.28, 11-inf 0.11}
```

To specify clock latency value for the clock-gated registers, use the `-stage` option with a value 0. Because you are specifying the latency value for the clock gated registers, the value for the `-fanout_latency` option should be 1-infinity, as shown in the following example:

```
set_clock_gate_latency -clock CLK -stage 0 \
-fanout_latency { 1-inf 0.1 }
```

For more details, see the command man page.

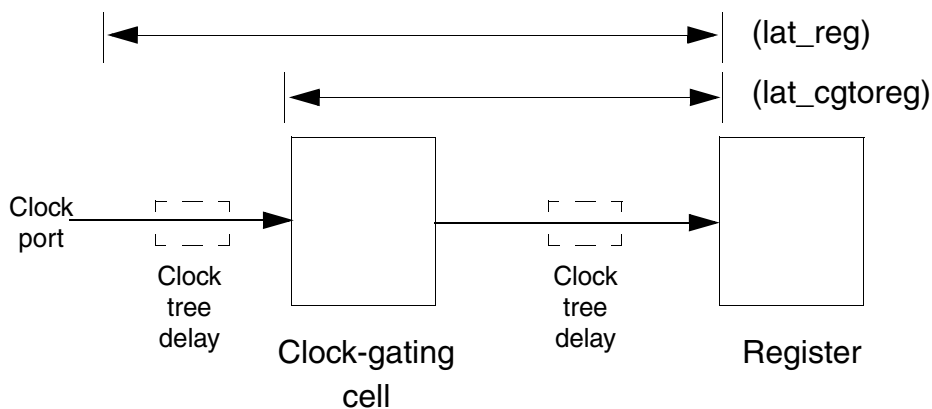
The set_clock_latency Command

In RTL designs, after you insert the clock gating cells use the `set_clock_latency` command to specify the clock network latency. Use this command only when your design already has clock-gating cells.

In the following example, shown in [Figure 7-4](#),

- `lat_cgtoreg` is the estimated delay from the clock pin of the clock gating cell to the clock pin of the gated register.
- `lat_reg` is the estimated clock-network latency to the clock pins of the registers without clock gating.

Figure 7-4 Clock Latency With Clock-Gating Design



For all clock pins of registers (gated or ungated) in the design that are driven by a particular clock, use the `lat_reg` value for the `set_clock_latency` command. For clock pins of all the clock-gating cells, use the value of `lat_reg-lat_cgtoreg` for the `set_clock_latency` command. Because the purpose of setting the latency values is to account for the different

clock-network delays between the registers and the clock-gating cell it is important to get a reasonably accurate value of the difference (`lat_cgtoereg`). The absolute values used are relatively less important, unless you are using these values to account for clock-network delay issues not related to clock gating.

For more details, see the command man page.

Applying Clock-Gate Latency

The clock latency specified using the `set_clock_gate_latency` command is annotated on the registers during the `compile_ultra -gate_clock` command when the clock-gating cells are inserted. However, if you modify the latency values on the clock gates after the compilation, you must manually apply the latency values on the existing clock-gating cells using the `apply_clock_gate_latency` command. This command can be used on the clock-gating cells inserted by the tool during the `compile_ultra -gate_clock` command or by the `insert_clock_gating` command.

Note:

Having modified the clock-gate latency using the `set_clock_gate_latency` command, if you compile your design using the `compile_ultra` or `compile_ultra -incremental` command, using the `apply_clock_gate_latency` command is not necessary. The tool annotates the specified value during the compilation.

For more details, see the command man pages.

Resetting Clock-Gate Latency

To remove the clock latency information specified on the clock-gating cells, use the `reset_clock_gate_latency` command. This command removes the clock latency values on the specified clocks. If you do not specify the clock, the clock latency values on all the clock-gating cells are removed. This command removes the clock latency on the specified clocks, irrespective of whether the latency values were specified using the `set_clock_latency` or the `set_clock_gate_latency` commands.

For more details, see the command man page.

Comparison of the Clock-Gate Latency Specification Commands

Table 7-2 compares various commands that you can use to specify the clock-gate latency.

Table 7-2 Comparison of Clock-Gating Latency Specification Commands

set_clock_gate_latency	set_clock_gating_style -setup -hold	set_clock_gating_check	set_clock_latency
Recommended to be used with the <code>compile_ultra -gate_clock</code> command	Default values are recommended for most designs. Use this command only if the default values are not suitable for your design	To specify the clock-gate latency on existing clock-gating cells.	To modify clock-gate latency on existing clock-gating cells.
To specify clock-gate latency before the clock gates are inserted by the <code>compile_ultra -gate_clock</code> command	If used before the <code>insert_clock_gating</code> command, requires you to use the <code>propagate_constraints</code> command after the clock-gate insertion. If used before the <code>compile_ultra -gate_clock</code> command, constraint propagation is automatically done after the clock-gate insertion.	Specification is on the instance. So, specify on each clock-gating cell.	Specification is on the instance. So, specify on each clock-gating cell
To modify the clock-gate latency settings on existing clock-gating cells	To specify the setup and hold values before the clock gates are inserted	Specification overrides the setup and hold values in the technology library	The latency setting specifies the clock arrival time at the clock-gating cell
The latency setting specifies the clock arrival time at the clock-gating cell	The specification overrides the setup and hold values defined in the technology library	Can be used with both <code>insert_clock_gating</code> and <code>compile_ultra -gate_clock</code> command	Can be used with both <code>insert_clock_gating</code> and <code>compile_ultra -gate_clock</code> command

Table 7-2 Comparison of Clock-Gating Latency Specification Commands (Continued)

set_clock_gate_latency	set_clock_gating_style -setup -hold	set_clock_gating_check	set_clock_latency
Specification is based on clock domain, clock-gating state and fanout	Generic settings for all the clock gates in the design		

Calculating the Clock Tree Delay From Clock-Gating Cell to Registers

If your clock tree synthesis tool does not insert buffers after the clock-gating cell, then the total delay between the clock-gating cell and the registers is equal to the delay of the clock-gating cell (clock pin to clock out signal) plus the wire delay between the clock-gating cell and the registers. If your clock tree synthesis tool inserts buffers after the clock-gating cell, add an estimate of the clock-network delay to the total delay between the clock-gating cell and the registers. You can use an estimate based on the fanout of the clock-gating cell and the driving capacity of typical clock tree buffers or use data from earlier designs.

For most designs, the enable signal arrives early and is not affected by clock-network delay issues. For late arriving enable signals, it is advisable to be conservative (high value) in the selection of the delay from the clock-gating cell to the registers. A low value may mean an enable signal which is unable to meet arrival time constraints at the clock-gating cell after the clock tree is inserted. However, a high value may over constrain the enable signal leading to higher area or power and ensures that the enable signal arrives in time at the clock-gating cell.

After placement and clock tree synthesis, you can back-annotate delay information by using the `set_propagated_clock` command to inform Design Compiler to use real delay data for the clock-network delay. For more information, see the Design Compiler documentation.

Specifying Setup and Hold

During insertion of clock gates, the setup and hold time that you specify defines the margins within which the enable signal (EN) must operate to maintain the integrity of the gated-clock signal.

The setup and hold values for the integrated clock-gating cell are specified in the technology library. The values specified in the technology library are honored by `compile_ultra -gate_clock` command during clock gate insertion. However, you can override these values in the following ways:

- Specifying the `-setup` and `-hold` options in `set_clock_gating_style` command. By doing so, all the clock gates in the design should have the setup and hold time that you specify.
- For the clock-gating cells already existing in your design, use the `set_clock_gating_check` command to specify a desired setup and hold time. You cannot use this command if the clock gates are inserted during the `compile_ultra -gate_clock` command.

You use the `report_timing -to` command to the enable pin of the clock-gating cell to verify that the new values are correct.

The following example uses the `set_clock_gating_style` command to specify the setup and hold values:

```
set_clock_gating_style \
    -max_fanout 16 \
    -positive_edge_logic integrated \
    -setup 6 \
    -hold 2
compile_ultra -gate_clock
# to validate the user-specified setup/hold time for
# integrated clock gating
report_timing -to clk_gate_out_top_reg/EN
```

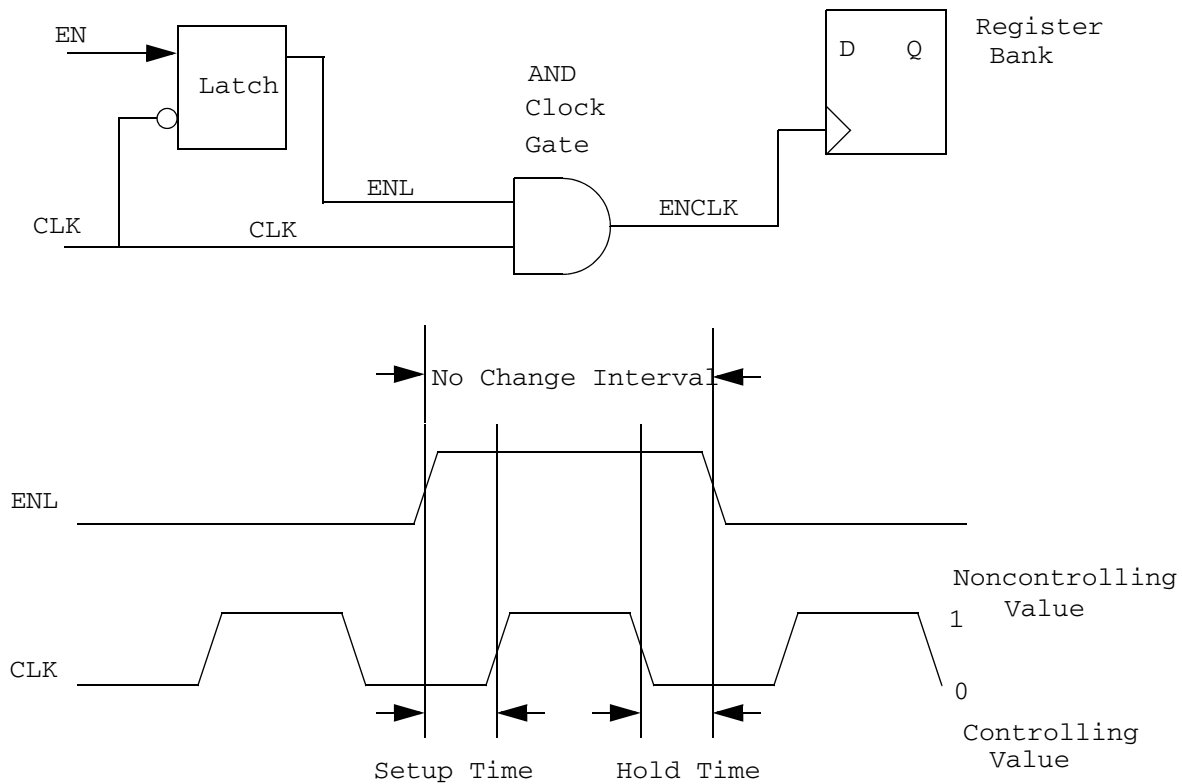
For example, using `set_clock_gating_check`:

```
set_clock_gating_style \
    -max_fanout 16 \
    -positive_edge_logic integrated \
    -control_point before \
    -control_signal test_mode
set_clock_gating_check -setup 3 -hold 2 [ get_cells
clk_gate_out_top_reg/main_gate ]
set_clock_gating_check -setup 5 -hold 1.5 [ get_cells
clk_gate_out_top_reg_1/main_gate ]
compile_ultra -gate_clock
# to validate the user-specified setup/hold time for
# integrated clock gating
report_timing -to clk_gate_out_top_reg/EN report_timing -to
clk_gate_out_top_reg_1/EN
```

The clock gate must not alter the waveform of the clock, other than turning the clock signal on and off. If the enable signal operates outside the properly chosen margins specified by `-setup` and `-hold`, the resulting gated signal can be clipped or otherwise corrupted.

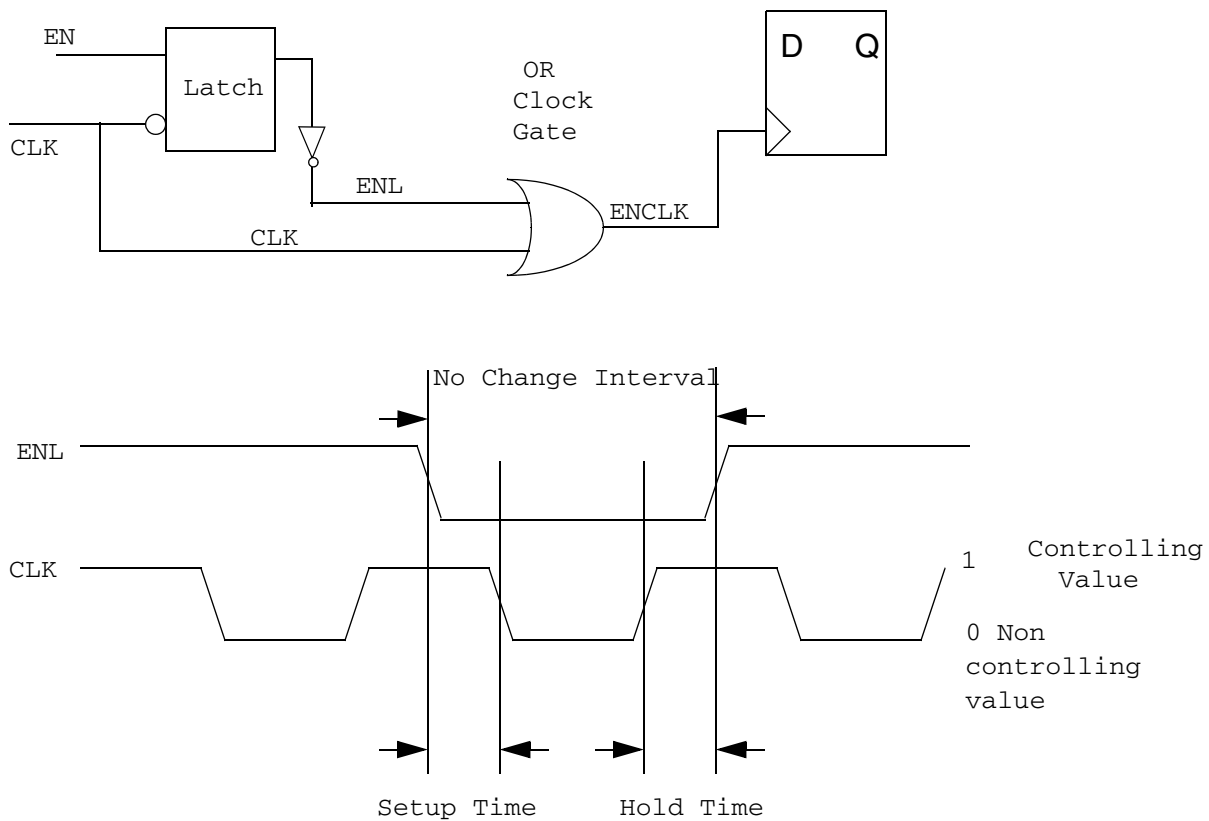
Figure 7-5 and Figure 7-6 show the relationship of setup and hold time to a clock waveform. Figure 7-5 shows the relationship with an AND gate as the clock-gating element. Figure 7-6 shows the relationship with an OR gate as the clock-gating element.

Figure 7-5 Setup and Hold Time for an AND Clock Gate



Enable after latch (ENL) signal must be stable before the clock input (CLK) makes a transition to a non-controlling value. The hold time ensures that the ENL is stable for the time you specify after the CLK returns to a controlling value. The setup and hold time ensures that the ENL signal is stable for the entire time that the CLK signal has a non-controlling value, which prevents clipping or glitching of the ENCLK clock signal.

You may need to add latency by using the `set_clock_latency` command. Use this command for non-clock-gating registers. For more information, see [“Specifying Clock-Gate Latency” on page 7-14](#) and the Design Compiler documentation.

Figure 7-6 Setup and Hold Time for an OR Clock Gate**Note:**

When using PrimeTime for static timing-analysis, use the `set_clock_gating_check -setup` and `-hold` options to change the setup and hold values for the gating check. PrimeTime performs clock-gating checks on all gated clocks using 0.0 as the default for setup and hold.

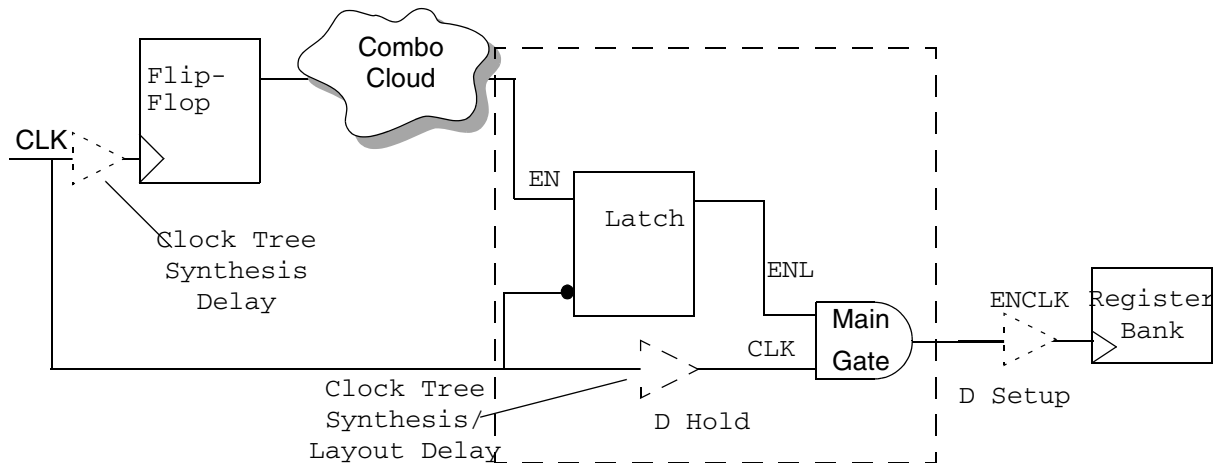
Predicting the Impact of Clock Tree Synthesis

Clock tree synthesis can affect your choice of setup and hold time. However, during clock gating, the clock tree does not exist yet: clock tree synthesis normally occurs much later in the design process than clock gating. Without the clock tree, it can be difficult to precisely predict the impact of clock tree synthesis on the delay of the design. For this reason, you might find it necessary to alter your setup and hold time after clock tree synthesis.

Choosing a Value for Setup

For `-setup` time, choose a value that estimates the delay impact of the clock tree from the clock gate to the gated register bank. In latch-based clock gating, the value for setup simply mimics the delay of the clock tree from the clock gate to the register bank.

Figure 7-7 Setup and Hold Time for Clock Tree Synthesis



Your setup time constrains the ENL signal so that after gate-level synthesis, there is still enough timing slack for the addition of the clock tree during clock tree synthesis.

In latch-free clock gating, the value for setup must consider the clock signal duty cycle. For example, in a design using a latch-free clock gate:

1. Estimate the delay of the clock tree between the clock gate and the gated register (as you would for the latch-based clock gate).
2. From the value you estimate in step 1, add the worst-case (largest possible) clock low time (typically half of the clock-cycle time).

This is appropriate for flip-flops triggered on the clock's rising edge. For flip-flops triggered on the clock's falling edge, add the worst-case (largest possible) clock high time.

If the value of `-setup` is too small, the ENL signal must be reoptimized after back-annotation from layout to fit the tighter timing constraints. If the value of `-setup` is too large, the ENL signal is too constrained and optimization of combinational control logic results in larger area and power to satisfy the tighter timing constraints.

Choosing a Value for Hold

Latch-based clock gating has the timing requirement that the transition of the ENL signal occur at the 2-input clock gate after the trailing edge (rising edge for falling-edge flip-flop) of the clock signal. This timing requirement is usually satisfied because clock gating's addition of a latch increases the delay on the ENL signal. In rare cases, however, after clock tree synthesis and physical design, additional delay in the clock signal might cause the CLK signal to arrive after the ENL signal. This is due to clock skew between the clock signal driving the clock-gating latch and the clock signal driving the 2-input gate.

If you expect this timing violation, you can set the `-hold` value during clock gating to artificially define a hold constraint on the ENL signal. Gate-level synthesis adds buffers in the ENL signal if they are necessary to satisfy your hold constraint.

If the value of `-hold` is too small, you might have to reoptimize the ENL signal after back-annotation from layout to ensure the integrity of the gated clock signal. If the value of `-hold` is too large, you might find a chain of buffers delaying the ENL signal before the clock gate.

Choosing Gating Logic

The following options of the `set_clock_gating_style` command specify the type of clock-gating logic or clock-gating cell used for implementing clock gating:

```
-positive_edge_logic [gate_list] [cell_list]
-negative_edge_logic [gate_list] [cell_list]
```

You can specify a configuration of 1- and 2-input gates (simple gating cells) to use for clock gating, or an integrated clock-gating cell already defined in the target library. An integrated cell is a dedicated clock-gating cell that combines all of the simple gating logic of a clock gate into one fully characterized cell, possibly with additional logic such as multiple enable inputs, active-low enabling logic, or an inverted gated clock output.

Choosing a Configuration for Gating Logic

The `-positive_edge_logic` and `-negative_edge_logic` options can have up to three string parameters that specify the type of clock gating logic:

- The type of 2-input clock gate (AND, NAND, OR, NOR)
- An inverter or buffer on the clock network before the 2-input clock gate
- An inverter or buffer on the clock network after the 2-input clock gate

The positions of the string parameters determine whether clock gating places a buffer or inverter before or after the 2-input clock gate. For example, if the value of `-positive_edge_logic` is `{and buf}`, clock gating uses an AND gate and places a buffer in the fanout from the AND gate. If the value is `{inv nor}`, clock gating uses a NOR gate and places an inverter in the fanin of the NOR gate. Both of these examples result in AND functionality of the clock gate.

The type of logic that is appropriate for gating your circuit depends on,

- Whether the gated register banks are inferred by rising- or falling-edge clock constructs in your HDL code
and
- Whether you use latch-based or latch-free clock gating

When using latch-free clock gating, you must specify both the `-positive_edge_logic` and `-negative_edge_logic` options.

For proper operation of the gated design, use the `-positive_edge_logic` and `-negative_edge_logic` options of the `set_clock_gating_style` command to choose any combination of gates that provides the appropriate functionality shown in [Table 7-3](#) and [Table 7-4](#). [Table 7-3](#) provides information for the latch-based clock-gating style. [Table 7-4](#) provides information for the latch-free clock-gating style.

Table 7-3 Gating Functionality for Latch-Based Clock Gating

Gating logic -pos{} or -neg{}	Latch-based clock gating			
	Rising-edge-triggered registers ¹		Falling-edge-triggered registers ²	
	Valid?	Remarks	Valid?	Remarks
{and}	Yes			
{or}			Yes	(³)
{nand}	Yes	Clock gating adds an inverter to the clock line to the register.		
{nor}			Yes	Clock gating removes the inverter from the clock line to the register.

Table 7-3 Gating Functionality for Latch-Based Clock Gating (Continued)

Gating logic -pos{} or -neg{}	Latch-based clock gating			
	Rising-edge-triggered registers ¹		Falling-edge-triggered registers ²	
	Valid?	Remarks	Valid?	Remarks
{and inv}	Yes	Clock gating adds an inverter to the clock line to the register.		
{or inv}			Yes	Clock gating removes the inverter from the clock line to the register.
{nand inv}	Yes			
{nor inv}			Yes	
{inv and}			Yes	Clock gating removes the inverter from the clock line to the register.
{inv or}	Yes	Clock gating adds an inverter to the clock line to the register.		
{inv nand}			Yes	(⁴)
{inv nor}	Yes			
{inv and inv}			Yes	(⁴)
{inv or inv}	Yes			
{inv nand inv}			Yes	Clock gating removes the inverter from the clock line to the register.
{inv nor inv}	Yes	Clock gating adds an inverter to the clock line to the register.		

1. If Power Compiler adds an inverter on the clock line to a rising-edge-triggered register, Design Compiler might infer a falling-edge-triggered register during later synthesis if one is available in your technology library. This is normal.

2. If Power Compiler removes an inverter from the clock line to a falling-edge-triggered register, Design Compiler might infer a rising-edge-triggered register if one is available in your technology library. This is normal.

3. The enable input of the OR gate has an inverter to ensure correct functionality when using clock gating.
4. The enable input of the OR gate has an inverter to ensure correct functionality when using clock gating. This cancels the effect of the additional inverter on the enable input signal. Therefore only the clock pin of the main gate is inverted.

Table 7-4 Gating Functionality for Latch-Free Clock Gating

Gating logic -pos{} or -neg{}	Latch-free clock gating			
	Rising-edge-triggered registers ¹		Falling-edge-triggered registers ²	
	Valid?	Remarks	Valid?	Remarks
{and}			Yes	
{or}	Yes	(³)		
{nand}			Yes	Clock gating removes the inverter from the clock line to the register.
{nor}	Yes	Clock gating adds an inverter to the clock line to the register.		
{and inv}			Yes	Clock gating removes the inverter from the clock line to the register.
{or inv}	Yes	Clock gating adds an inverter to the clock line to the register.		
{nand inv}			Yes	
{nor inv}	Yes	(³)		
{inv and}	Yes	Clock gating adds an inverter to the clock line to the register.		
{inv or}			Yes	Clock gating removes the inverter from the clock line to the register.
{inv nand}	Yes	(⁴)		

Table 7-4 Gating Functionality for Latch-Free Clock Gating (Continued)

Gating logic -pos{} or -neg{}	Latch-free clock gating			
	Rising-edge-triggered registers ¹		Falling-edge-triggered registers ²	
	Valid?	Remarks	Valid?	Remarks
{inv nor}			Yes	
{inv and inv}	Yes	(⁴)		
{inv or inv}			Yes	
{inv nand inv}	Yes	Clock gating adds an inverter to the clock line to the register.		
{inv nor inv}			Yes	Clock gating removes the inverter from the clock line to the register.

1. If Power Compiler adds an inverter on the clock line to a rising-edge-triggered register, Design Compiler might infer a falling-edge-triggered register during later synthesis if one is available in your technology library. This is normal.
2. If Power Compiler removes an inverter from the clock line to a falling-edge-triggered register, Design Compiler might infer a rising-edge-triggered register if one is available in your technology library. This is normal.
3. The enable input of the OR gate has an inverter to ensure correct functionality when using clock gating.
4. The enable input of the OR gate has an inverter to ensure correct functionality when using clock gating. This cancels the effect of the additional inverter on the enable input signal. Therefore only the clock pin of the main gate is inverted.

For example, to achieve AND functionality, you can simply use an AND gate. However, AND functionality also results from the combination of an INV and a NOR gate. Any combination of individual gates is allowable if the combination results in the appropriate functionality shown in [Table 7-3](#) and [Table 7-4](#).

In the following example, latch-based clock gating uses an AND gate for gating clocks of rising-edge-triggered register banks and an OR gate for gating clocks of falling-edge-triggered register banks. The enable input of the OR gate has an inverter to ensure correct functionality when using clock gating.

```
-positive_edge_logic {and} -neg {or}
```

In the following example, latch-based clock gating chooses a NOR gate for gating clocks of rising-edge-triggered register banks. Clock gating inserts an inverter in the fanin to the 2-input clock gate and a buffer in the fanout from the 2-input clock gate. This combination results in AND functionality.

```
-positive_edge_logic {inv nor buf} -neg {inv and inv}
```

For falling-edge-triggered register banks in this example, clock gating uses an AND gate to gate the clock. Clock gating inserts inverters in the fanin and fanout of the 2-input clock gate. This combination results in OR functionality. The enable input of the OR gate already has an inverter. This cancels the effect of the additional inverter on the enable input signal. Therefore, only the clock pin of the main gate is inverted.

Choosing a Simple Gating Cell by Name

The syntax of the `-positive_edge_logic` and `-negative_edge_logic` options allows you to use a specific clock-gating cell during clock gating. To use a specific gating cell from the target library, enter the cell name after the element type, separating the two with a colon.

In the following example for rising-edge-triggered register banks, latch-based clock gating chooses the specific AND gate, MYAND2, from the target library. In this example, clock gating inserts a buffer in the fanout of the clock gate.

```
-positive_edge_logic {and:MYAND2 buf}
```

Choosing a Simple Gating Cell and Library by Name

In some cases, you might have more than one target library with cell names that are the same. In such cases, you can use a specific cell from a specific library for clock gating. The syntax of `-positive_edge_logic` and `-negative_edge_logic` allows you to indicate a specific library and cell for clock gating, as follows.

```
target_library = { "CMOS8_MAX.db" "tech_lib1.db"
                  "tech_lib2.db" }

-positive_edge_logic {and:tech_lib1/MYAND2 buf:tech_lib2/
MYBUF2}
```

In this example, clock gating uses a particular AND cell and BUF cell from different technology libraries. The AND cell is MYAND2 from the tech_lib1 library, and the buffer is MYBUF2 from the tech_lib2 library. You must have previously specified these technology libraries as target libraries by setting the Design Compiler `target_library` variable.

Choosing an Integrated Clock-Gating Cell

You can use the `-positive_edge_logic` and `-negative_edge_logic` options of the `set_clock_gating_style` command to specify the integrated clock-gating cell for clock gating:

```
-positive_edge_logic [gate_list] [cell_list]
-negative_edge_logic [gate_list] [cell_list]
```

If the integrated clock-gating cell is a generic type cell, you can use the `-gicg_*` options of the `set_clock_gating_style` command, as described in [“Choosing a Generic Integrated Cell” on page 7-31](#).

The first cell found that meets the clock-gating requirements is used and possibly sized up or down to meet the design rule violations if the library has integrated cells of different sizes. As desired, use the `power_do_not_size_icg_cells` variable to prevent this behavior.

Choosing an Integrated Cell by Functionality

When selecting an integrated cell by functionality, clock gating searches your technology library for integrated cells having the correct value of the `clock_gating_integrated_cell` attribute.

Use the `set_clock_gating_style` command to specify the functionality of the integrated cell you want clock gating to look for.

Power Compiler uses the first integrated cell it finds in your library that matches the requirements you specify with the `set_clock_gating_style` command. For example, if you enter

```
set_clock_gating_style -neg {integrated}
```

Power Compiler uses the first integrated cell it finds in your technology library that has the `clock_gating_integrated_cell` attribute, as follows:

```
clock_gating_integrated_cell : "latch_negedge";
```

You do not need to specify latch-based or latch-free gating if you use the default latch-based gating. For more information about attributes for integrated cells and Library Compiler syntax, see the Library Compiler documentation.

Choosing an Integrated Cell by Name

Choose an integrated cell by name when you require a specific integrated cell or if you have more than one integrated cell with the same `clock_gating_integrated_cell` attribute. For example,

```
set_clock_gating_style -pos {integrated:my_cell}
```

In this example, clock gating chooses an integrated cell called `my_cell` from the technology library. For more information about attributes for integrated cells and Library Compiler syntax, see the Library Compiler documentation.

Choosing a Generic Integrated Cell

To use generic integrated clock-gating cells for implementing clock-gating functions, specify the desired cells by using the `set_clock_gating_style` command with the following options:

```
-gicg_pos_cell gicg_libcell
-gicg_neg_cell gicg_libcell
-gicg_pos_auto base_libcell
-gicg_neg_auto base_libcell
```

Use `-gicg_pos_cell` or `-gicg_neg_cell` (or both) to specify the exact library cell to be used for clock gating. Use `-gicg_pos_auto` or `-gicg_neg_auto` (or both) to have Power Compiler automatically select the best generic integrated clock-gating cell to use. For more information, see [“Generic Integrated Clock-Gating Cells” on page 7-46](#).

Specifying a Subset of Integrated Clock Gates for Clock-Gate Insertion

Use the `set_dont_use -power` command to limit clock gate insertion to a specific set of integrated clock gate cells from one or more libraries. This command guarantees that the specified cells is not used for power optimization. For example,

```
set_dont_use -power [get_lib_cell a1.db/icg_a1_*]
set_dont_use -power [get_lib_cell b2.db/icg_b2_*]
set_dont_use -power [get_lib_cell c3.db/icg_c3_*]
set_clock_gating_style -pos {integrated}
compile_ultra -gate_clock
```

In the example mentioned above, the `set_clock_gating_style` command directs the `compile_ultra -gate_clock` command to use all integrated cells except for those that have the don't use attribute.

Using Setup and Hold for Integrated Cells

Setup and hold constraints are built into the integrated cell when you create it with Library Compiler, but you can override the values by using either the `set_clock_gating_style` command or the `set_clock_gating_check` command.

If you provide `-setup` and `-hold` values on the command line when using an integrated cell, the values are overridden.

Consider the following example that uses an integrated cell to gate rising-edge-triggered registers and uses simple cells to gate falling-edge-triggered registers using latch-free style.

Example

```
set_clock_gating_style -seq none
-setup setup_value
-hold hold_value
-pos {integrated}
-neg {inv nor buf}
```

The `setup_value` and `hold_value` apply not only to the integrated cell, but also to the clock gate built for falling-edge-triggered registers using simple cells (INV, NOR, and BUF gates in this example). For more information about integrated cells and timing, see the Library Compiler documentation.

Designating Simple Cells Exclusively for Clock Gating

During technology mapping, Design Compiler builds clock-gating logic, using the generic representation created by Power Compiler and cells from your technology library.

Unless you are using an integrated cell for gating, there is nothing to prevent Design Compiler from using the same cells for mapping other parts of the design.

You can designate certain cells to be used exclusively or preferentially for gating clocks. Such cells can be the 2-input clock gate, inverters, buffers, or latches used in the latch-based style of clock gating.

To use a specific cell for clock gating and preclude its use in other areas of the design, set the following Library Compiler attributes to `true` in the library description of the cell:

- `dont_use`

When set to `true`, this attribute prevents Design Compiler from choosing the cell when mapping the design to technology.

- `is_clock_gating_cell`

This is an attribute of type Boolean for the cell group. When set to `true`, this attribute identifies the cell for use in clock gating. If `dont_use` and `is_clock_gating_cell` are both set to `true`, the cell is used only in clock-gating circuitry.

You can set `dont_use` and `is_clock_gating_cell` on

- 2-input clock gates

Examples of 2-input clock gates are AND, NAND, OR, and NOR library cells that are used to gate clocks.

- 1-input clock gates

Examples of 1-input clock gates are buffer and inverter library cells that are used in the fanin and fanout of the 2-input clock gate.

- 2-input D latches

These latches can be active high or low and must have a noninverting output.

To use a cell preferentially in clock gating, set only the `is_clock_gating_cell` attribute to `true`. Clock gating uses such cells preferentially when inserting clock-gating circuitry. Later, Design Compiler can use them as well when mapping other parts of the design to the target technology.

For more information about the syntax and use of Library Compiler attributes, see the Library Compiler documentation.

The 2-input clock gate has an enabling input and a clock input that is connected to ENL and CLK signals in [Figure 7-2](#). If the clock attribute is set on one of the pins of the 2-input clock gate, Power Compiler recognizes the remaining input pin as the enable pin. However, library cell syntax allows you to explicitly designate an input pin as the enabling input. In the pin group of the library description for the cell, set the `clock_gate_enable_pin` attribute to `true`. This is an attribute of type Boolean for the pin group.

Example

```
clock_gate_enable_pin : true;
```

If Power Compiler finds neither a clock attribute nor a `clock_gate_enable_pin` attribute, the software checks for the existence of setup and hold time on the pins. If setup and hold time are found on a pin, the software uses that pin as the enable pin. For more information about Library Compiler syntax and cell descriptions, see the Library Compiler documentation.

Selecting Clock-Gating Style

Use the `set_clock_gating_style` command to select options for clock-gating style. The options you select are implemented when clock gating cells are inserted by the `compile_ultra -gate_clock` and the `insert_clock_gating` commands. The default values of the `set_clock_gating_style` command is suitable for most designs. If the default settings does not suite your design use this command to change the default settings.

With the `set_clock_gating_style` command you can,

- Select a latch-based or latch-free clock-gating style (the default is latch-based, with or without an integrated cell)

- Assign values for setup and hold times at the enable input of the clock-gating cell (the default is 0)
- Determine how much test logic the tool adds during clock gating (the default is none) to improve controllability and observability
- Specify generic integrated clock-gating cells

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

Choosing a Specific Latch and Library

The `-sequential_cell` option allows you to use a specific latch when inserting clock-gating circuitry. To use a specific latch from the target library, enter the name of the latch after the element type, separating the two with a colon (:). For example:

```
-sequential_cell latch:LAH10
```

To designate a specific latch from a specific target technology library, insert the name of the technology library as shown in the following example. Clock gating uses a latch called LAH10 from the target library.

```
-sequential_cell latch:SPECIFIC_TECHLIB/LAH10
```

In the next example, clock gating uses the LAH10 latch from a technology library called SPECIFIC_TECHLIB. You must previously have specified this technology library file name when setting the Design Compiler `target_library` variable.

```
target_library = { "CMOS8_MAX.db" "SPECIFIC_TECHLIB.db" }
```

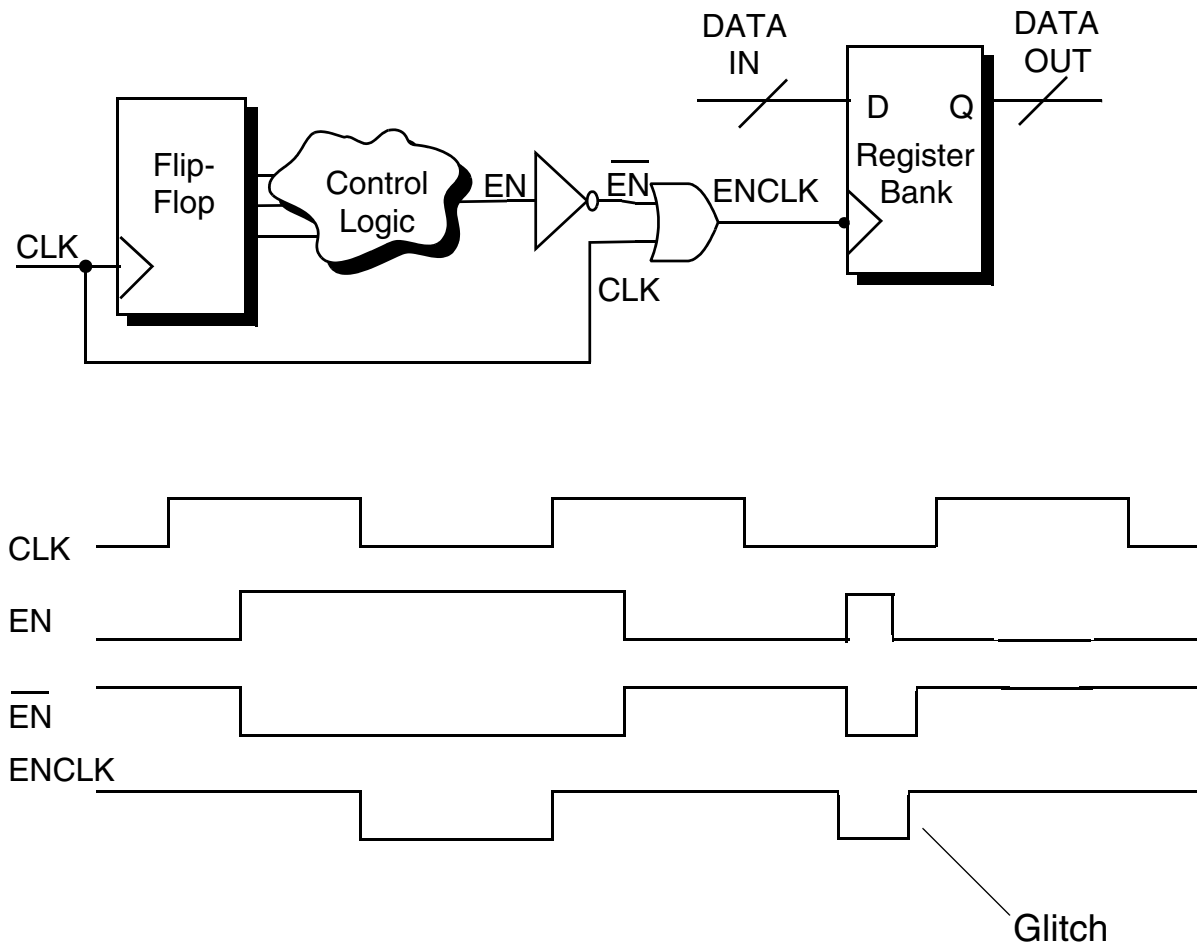
Note:

By convention, the technology library name and file name are usually different. For example, CMOS8_max is the name of a technology library. The file name for it can be any name.lib. The .lib extension means the library is in Liberty text format. In this example, CMOS8_MAX.lib is the file name for this library in text format. CMOS8_MAX.db is the file name for this library in Synopsys proprietary binary format.

Choosing a Latch-Free Style

The `-sequential_cell` option of `set_clock_gating_style` command allows you to select a clock-gating style that uses latches or avoids the use of latches. [Figure 7-2](#), earlier in this chapter, shows an example of the latch-based clock-gating style. An example of a circuit with the latch-free clock-gating style is shown in [Figure 7-8](#).

Figure 7-8 Latch-Free Clock Gating



In this example of the latch-free style, clock pulses to the register bank are gated by the OR gate. In the latch-free style, the clock gate prevents the trailing clock edge. Figure 7-8 shows a latch-free clock gate for rising-edge-triggered logic. A latch-free clock gate for rising-edge-triggered logic prevents the falling clock edge.

Eliminating the latch can reduce power dissipation and area slightly. However, the latch-free method has a significant drawback: The EN signal must be stable at its new value before the falling clock edge. If the EN signal is not stable before the falling clock edge, glitches on the EN signal can corrupt the clock signal to the register. Any glitches on the EN signal after the trailing edge of the clock lead to glitching and corruption of the gated clock signal. See Figure 7-8 for an example of latch-free clock gating.

Improving Testability

Clock gating introduces multiple clock domains in the design. Introducing multiple clock domains can affect the testability of your design unless you add logic to enhance testability.

In certain scan register styles, a gated register cannot be included in a scan chain, because gating the register's clock makes it uncontrollable for test (assuming there is no dedicated scan clock). Without the register in the scan chain, test controllability is reduced at the register output and test observability is reduced at the register input. If you have many gated registers, this can significantly reduce the fault coverage in your design.

You can improve the testability of your circuit by using the options of the `set_clock_gating_style` command to determine the amount and type of testability logic added during clock gating. You can perform the following steps to improve testability:

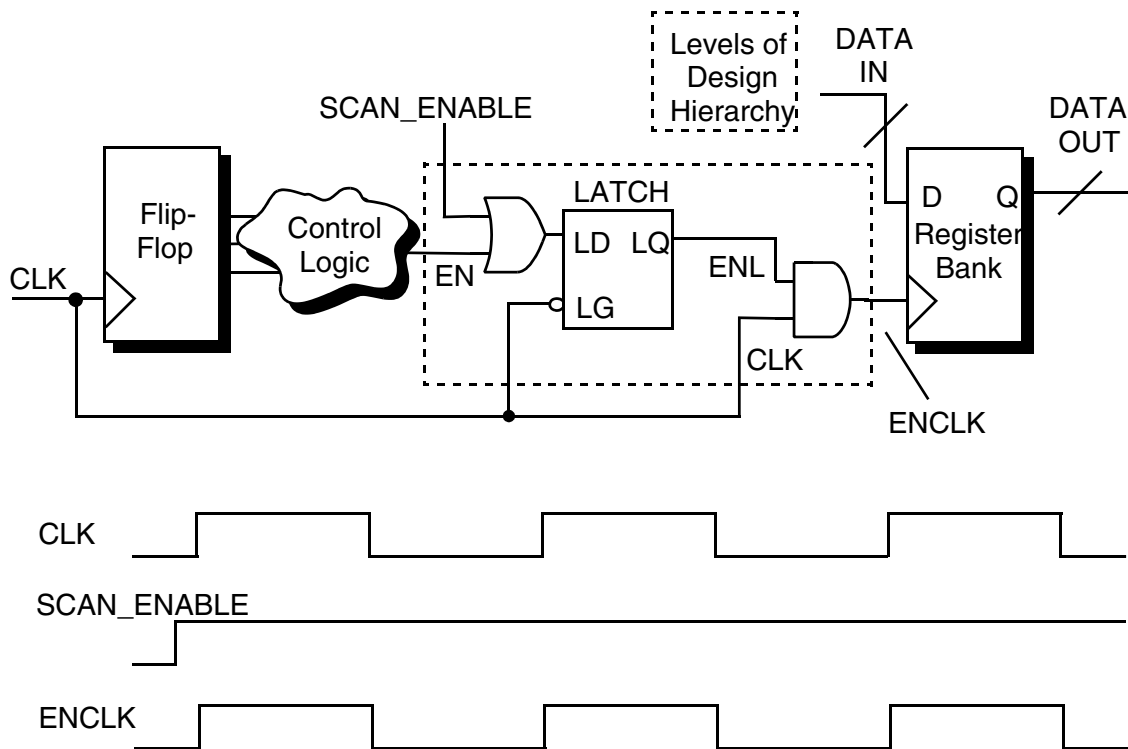
- Add a control point for testing
- Choose `test_mode` or `scan_enable`
- Add observability logic

Inserting a Control Point for Testability

A control point increases the testability of your design by restoring the clock signal to its ungated form during test. The control point is an OR gate that eliminates the function of the clock gate during test, which restores the controllability of the clock signal.

[Figure 7-9](#) shows a control point (OR gate) connected to the `scan_enable` port. The control point is before the latch in this example.

Figure 7-9 Control Point in Gated Clock Circuitry



When the scan_enable signal is high, the test signal overrides clock gating, thus making the ENCLK and CLK signals identical during shift mode. The test solution in [Figure 7-9](#) has the advantage of achieving testability with the addition of only one OR gate. This configuration has fault coverage comparable to that of a design without clock gating.

The `set_clock_gating_style` command has two options to determine the location and type of the control point for test:

- `-control_point none | before | after`

The default is `none`. The `-control_point` option inserts your control point before or after the clock-gating latch. When using the latch-free clock-gating style, before and after are equivalent.

- `-control_signal test_mode | scan_enable`

The default is `scan_enable`. This option creates a `scan_enable` or `test_mode` test port and connects the port to the control-point OR gate. DFT Compiler interprets `test_mode` and `scan_enable` in a specific manner. The `-control_signal` option also applies to any observability logic inserted by the `-observation_point` option. You can use the `control_signal` option only if you have used the `-control_point` option.

When creating the control point, Power Compiler creates and names a new test port and assigns appropriate attributes to the port. [Table 7-5](#) shows variables that Power Compiler checks when naming the new port and when setting attributes on it.

Table 7-5 Test Port Naming and Attribute Assignment

Setting of -control_signal	Variable that determines test port name	Attributes on test port are the same as those set by
scan_enable	test_scan_enable_port_naming_style	set_dft_signal -type ScanEnable
test_mode	test_mode_port_naming_style	set_attribute test_port_clock_gating set_dft_signal -type TestMode

To connect the test port of the clock-gating design to the test port of your design, use the `insert_dft` command. For more information, see [“Connecting the Test Ports Throughout the Hierarchy” on page 7-41](#).

Latch-based clock gating requires that the enable signal always arrive after the trailing edge (rising edge for falling-edge signal) of the clock. If you insert the control point before the latch, it is impossible for the control point to violate this requirement. However, your test tool might not support positioning the control point before the clock-gating latch. In such cases, use `-control_point after` to insert the control point after the clock-gating latch.

Note:

If you insert the control point after the latch, the `scan_enable` signal or `test_mode` signal must transition after the trailing edge (rising edge for falling-edge signal) of the clock signal during test at the foundry; otherwise glitches in their resulting signal corrupts the clock output.

Scan Enable Versus Test Mode

Scan enable and test mode differ in the following way:

- Scan enable is active only during scan mode.
- Test mode is active during the entire test (scan mode and parallel mode).

Scan enable typically provides higher fault coverage than test mode. Fault coverage with scan enable is comparable to a circuit without clock gating. However, there can be situations in which you must use test mode. For example, you might need to use test mode if you place the control point before the latch and your test tool does not support this position of the control point with scan enable.

Improving Observability With `test_mode`

When using test mode, the EN signal and other signals in the control logic are untestable. If your test methodology requires that you use `test_mode`, you might need to increase your fault coverage. You can increase fault coverage with test mode by adding observability logic during clock gating.

Note:

When using `-control_signal scan_enable`, increasing observability with observability logic is not necessary.

The `set_clock_gating_style` command has two options for increasing observability when using `-control_signal test_mode`:

- `-observation_point true | false`

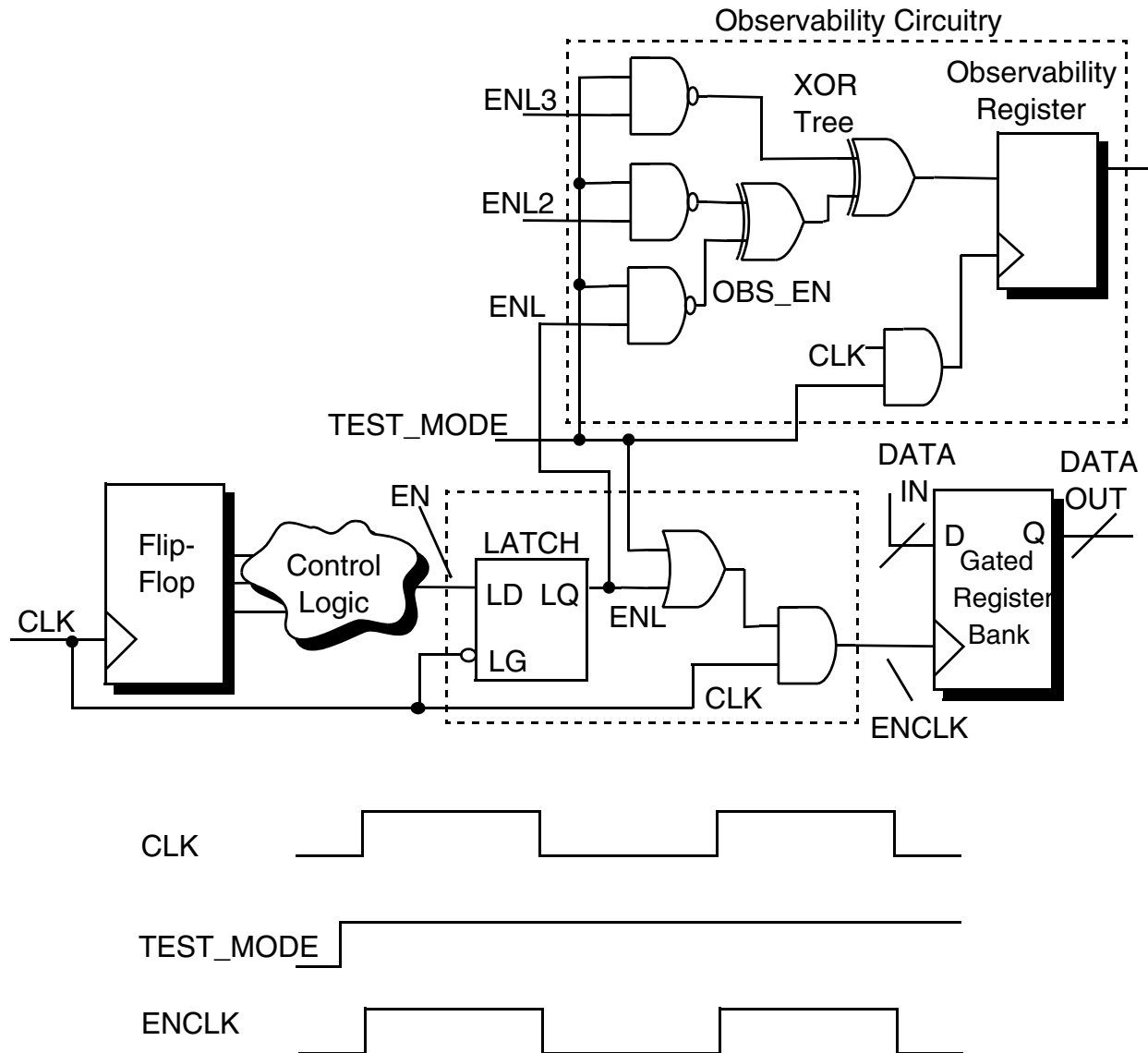
The default is `false`. When you set this option to `true`, clock gating adds a cell that contains at least one observability register and an appropriate number of XOR trees (if there is only one signal to be observed, an XOR tree is unnecessary). The scan chain includes the observability register, but the observability register's output is not functionally connected to the circuit.

- `-observation_logic_depth depth_value`

The default is 5. The value of this option determines the depth of logic of the XOR tree that `-observation_point` builds during clock gating. If this value is set to 0, each ENL signal is latched separately and no XOR tree is built. The XOR tree reduces the number of observability registers needed to capture the test signature.

[Figure 7-10](#) shows a gated clock, including an observability register and an XOR tree.

Figure 7-10 Gated Clock With High Observability



During test, observability circuitry allows observation of the ENL signal. During normal operation of the circuit, the XOR tree does not consume power, because the NAND gate blocks all ENL signal transitions. This test solution has high testability and is power-efficient, because the XOR tree consumes power only during test and the clock of the observability register is gated.

To connect the test port of the clock-gating design to the test port of your design, see [“Connecting the Test Ports Throughout the Hierarchy” on page 7-41](#).

Choosing a Depth for Observability Logic

Use the `-observation_logic_depth` option of the `set_clock_gating_style` command to set the logic depth of the XOR tree in the observability cell. The default for this option is 5.

Power Compiler builds one observability cell for each clock-gated design. Each gated register in the design provides a gated enable signal (OBS_EN in [Figure 7-10](#)) as input to the XOR tree in the observability cell.

If you set the logic depth of your XOR tree too small, clock gating creates more XOR trees (and associated registers) to provide enough XOR inputs to accommodate signals from all the gated registers. Each additional XOR tree adds some overhead for area and power. Using one XOR tree adds the least amount of overhead to the design.

If you set the logic depth of your XOR tree too high, clock gating can create one XOR tree with plenty of inputs. However, too large a tree can cause the delay in the observability circuitry to become critical.

Use the following guidelines in choosing or changing the logic depth of your XOR tree. Choose a value that is

- High enough to cause the construction of as few XOR trees as possible and
- Low enough to keep the delay in the observability circuitry from becoming critical

Connecting the Test Ports Throughout the Hierarchy

You use the `insert_dft` command to connect the test ports through various level of the design hierarchy.

If you have used the clock-gating feature of Power Compiler with the testability options, you must connect the test ports using the `insert_dft` command. After you have compiled all the lower level hierarchies of the design, use the command on the top level of the design.

There are two types of test ports: the `test_mode` port and the `scan_enable` port. A port can be recognized as a test port if it is designated as a `scan_enable` or a `test_mode` port using the `set_dft_signal` command. Alternatively, a port can be designated as a test port by setting the `test_port_clock_gating` attribute on it.

A `scan_enable (test_mode)` port is only connected to other `scan_enable (test_mode)` ports in the design hierarchy. If a `scan_enable (test_mode)` port exists at a particular level of the hierarchy, it is connected to `scan_enable (test_mode)` ports at all higher levels of the hierarchy. If a `scan_enable (test_mode)` port does not exist at a higher level of hierarchy, the `scan_enable (test_mode)` port is created.

The `insert_dft` command connects the test ports on all levels of the design hierarchy to the `test_mode` or `scan_enable` pins of the OR gate in the clock gating logic and the XOR gates in the clock-gating observability logic. If the design does not have a test port at any level of hierarchy, a test port is created. If a test port exists, it is used.

Using the `insert_dft` Command

You use the `insert_dft` command to connect the top-level test ports to the test pins of the clock-gating cells through the design hierarchy. A test port is created if the design does not have a test port at any level of the hierarchy. To identify the test ports, the tool uses the options you specified using the `set_dft_signal` command. The following example shows the usage of the `insert_dft` command to connect to the clock-gating cells. When you specify the value `clock_gating` to the `-usage` option of the `set_dft_signal` command, during the execution of the `insert_dft` command, the tool connects the specified signal to the test pin of the clock-gating cells.

```
dc_shell> read_ddc design.ddc
dc_shell> set_clock_gating_style -control_signal scan_enable \
    -control_point before
dc_shell> compile_ultra -scan -gate_clock
dc_shell> set_dft_signal -type ScanEnable -port test_se_1
dc_shell> set_dft_signal -type ScanEnable -port test_se_2 \
    -usage clock_gating
dc_shell> create_test_protocol
dc_shell> dft_drc -verbose
dc_shell> preview_dft
dc_shell> insert_dft
```

For more information, see *DFT Compiler Scan User Guide*.

Default Clock-Gating Style

The `compile_ultra -gate_clock` and the `insert_clock_gating` command honor the clock-gating style that you choose with the `set_clock_gating_style` command. The default values of the `set_clock_gating_style` command is suitable for most designs. If the default settings do not suite your design use this command to change the default settings.

If the clock-gating style is not chosen or set, Power Compiler looks for the integrated cells in the target library in the order shown below. User-specified clock-gating style always has priority.

- (a) `set_clock_gating_style -pos integrated -neg integrated \`
`-ctrl_pt before -ctrl_sig scan_enable`
- (b) `set_clock_gating_style -pos integrated -neg integrated \`
`-ctrl_pt after -ctrl_sig scan_enable`

- (c) `set_clock_gating_style -pos integrated -neg integrated \`
`-ctrl_pt before -ctrl_sig test_mode -obs_pt true`
- (d) `set_clock_gating_style -pos integrated -neg integrated \`
`-ctrl_pt after -ctrl_sig test_mode -obs_pt true`
- (e) `set_clock_gating_style -pos integrated -neg integrated`
- (f) `set_clock_gating_style -pos integrated -neg or \`
`-ctrl_pt after -ctrl_sig scan_enable`
- (g) `set_clock_gating_style -pos integrated -neg or \`
`-ctrl_pt before -ctrl_sig test_mode -obs_pt true`
- (h) `set_clock_gating_style -pos integrated -neg or \`
`-ctrl_pt after -ctrl_sig test_mode`
- (i) `set_clock_gating_style -pos integrated -neg or \`
`-ctrl_pt after -ctrl_sig test_mode -obs_pt true`
- (j) `set_clock_gating_style -pos integrated -neg or`
- (k) `set_clock_gating_style -pos and -neg integrated \`
`-ctrl_pt before -ctrl_sig scan_enable`
- (l) `set_clock_gating_style -pos and -neg integrated \`
`-ctrl_pt after -ctrl_sig scan_enable`
- (m) `set_clock_gating_style -pos and -neg integrated \`
`-ctrl_pt before -ctrl_sig test_mode -obs_pt true`
- (n) `set_clock_gating_style -pos and -neg integrated \`
`-ctrl_pt after -ctrl_sig test_mode -obs_pt true`
- (o) `set_clock_gating_style -pos and -neg integrated`
- (p) `set_clock_gating_style -pos and -neg or`

The following series of commands inserts clock gating by choosing the best default style:

```
dc_shell> read_verilog low.v
dc_shell> compile_ultra -gate_clock
dc_shell> report_clock_gating -style
dc_shell> report_power
dc_shell> insert_dft
dc_shell> compile_ultra -incremental (optional)
```

Modifying the Clock-Gating Structure

While performing RTL clock gating, you can specify the `set_clock_gating_style -max_fanout` option to limit the number of registers that are gated by a single clock-gating element. The results can be multiple clock-gating elements that have the same enable signal and, logically, the same gated-clock signal. All clock-gating cells with the same enable signal belong to the same clock-gating group. All registers gated by a single clock-gating element belong to the same clock-gating subgroup.

The gated registers inserted by the `compile_ultra -gate_clock` command are partitioned into subgroups. These partitions are not based on timing or placement constraints. So the placement tool tries to place the clock-gated registers close to the clock-gating cell, but this may not happen because of other design constraints. The result is a suboptimal partition of gated registers into subgroups.

You can correct this problem by moving clock-gated registers between the clock-gating cells belonging to the same clock-gating group. Because these clock-gating cells are logically equivalent, the rewired circuit is functionally valid.

Using the `rewire_clock_gating` command and `remove_clock_gating` command, you can rewire or remove clock gating in your design.

Changing a Clock-Gated Register to Another Clock-Gating Cell

The `rewire_clock_gating` command enables you to selectively rewire a clock-gated register from one clock-gating cell to another logically equivalent clock-gating cell.

You can use the `-undo` option to remove any rewiring you specified with the `rewire_clock_gating` command. Based on the options specified, the `-undo` option deletes the directives specified by the previously specified `rewire_clock_gating` command. Use the `-undo` option before you use the `compile -incremental` command. The `compile` command modifies the netlist to rewire the gated registers.

Because rewiring the gated registers alters the clock-gating cell that gates the registers, any path-based timing exception that goes through the old clock-gating cell to a gated register is no longer relevant and is lost.

Removing Clock Gating From the Design

Power Compiler performs clock gating at the RTL level during the compilation process when you use the `compile_ultra -gate_clock` command. The `remove_clock_gating` command lets you selectively remove the clock gates without having to start at RTL again.

The subsequent `compile_ultra` command removes the selected clock-gating cells. As a result you have the ability to use aggressive clock-gating strategies initially and selectively remove clock gating if needed.

This command also removes redundant clock-gating cells that are no longer connected to any clock-gating cells. Any associated test observation logic is also optimized.

All the registers that are ungated are remapped to new sequential cells. This may result in new pin names for the registers. If there were pin-based timing exceptions (by means of the `set_max_delay`, `set_min_delay`, `set_multicycle_path`, and `set_fast_path` commands) set on the pins of the old register, they may not be transferred properly during the transformation if the new and old pin names do not match.

The `remove_clock_gating` command displays a warning if there are pin-based timing exceptions on the register to be ungated. Cell based timing exceptions are not affected because the ungated registers retain their name. It is advisable to use the cell-based timing exceptions with clock-gating registers. For information, see the Design Compiler documentation.

Rewiring Clock Gating After Retiming

Power Compiler supports the `-balance_fanout` option to the `rewire_clock_gating` command.

This command is used to rebalance the fanout of the clock gates within the design after modifications have been made during retiming. During elaboration, Power Compiler automatically balances the register banks based on the minimum and maximum fanout requirements. However, when you issue commands such as `compile -ungroup` or `optimize` registers that perform retiming, registers can be removed if they are not loaded or to improve the timing. For clock-tree synthesis, it is important the clock gates have equivalent fanout loads: hence, the `-balance_fanout` option.

You use the `rewire_clock_gating -balance_fanout` command either after retiming or after compilation to restore a balanced fanout. When you use this command, Power Compiler compares the changed fanout of each equivalent clock-gating cell. The registers are moved around so that each equivalent clock-gating cell now has a balanced set of registers and honors the `-max_fanout` option that you specified originally. Any register banks not meeting the `min_bandwidth` requirement are ungated.

Note:

The command is not intended for use after the `balance_registers` command. When performing clock gating, it is recommended that you use the `optimize_registers` command.

Integrated Clock-Gating Cells

An integrated clock-gating cell integrates the various combinational and sequential elements of a clock gate into a single cell located in the technology library. An integrated clock-gating cell is a cell that you or your library developer creates to use especially for clock gating.

Consider using an integrated clock-gating cell if you are experiencing timing problems, such as clock skew, caused by the placement of clock-gating cells on your clock line.

Use Library Compiler to create an integrated cell for clock gating. For detailed information, see the Library Compiler documentation.

Library Compiler assigns a black box attribute to the complex sequential cells such as integrated clock-gating cells. Design Compiler does not use the integrated cells for the general logic synthesis. Power Compiler uses these integrated clock-gating cell for clock-gating. The selection of the clock-gating cell is controlled by the default or the user-specified values of the `set_clock_gating_style` command. Each integrated clock-gating cell in the library must contain the Library Compiler attribute called `clock_gating_integrated_cell`. This attribute can be set to either the string `generic` or to one of 26 strings that represent specific clock-gating types. The string `generic` causes Library Compiler to determine the functionality of the clock-gating cell, and allows Power Compiler to optimize the choice of clock-gating cells from the target library. Using one of the 26 standard strings specifies the functionality explicitly according to established conventions. For more details, see [Appendix A, "Integrated Clock-Gating Cell Example](#).

Generic Integrated Clock-Gating Cells

A “generic” integrated clock-gating cell does not specify the functionality of the cell explicitly in the `clock_gating_integrated_cell` attribute. Instead, Power Compiler determines the cell functionality and chooses the best cell to use.

A generic clock-gating cell can incorporate any kind of logic such as multiple enable inputs, test clock input, global scan input, asynchronous rst latch, active-low enabling logic, or inverted gated clock output. The creator of a generic clock-gating cell is not restricted to the 26 predefined types.

Power Compiler is free to optimize the enabling logic surrounding a generic integrated clock-gating cell by absorbing the surrounding logic into the logic functions available inside the cell, as long a compatible generic integrated clock-gating cell is available in the target library.

To use generic integrated clock-gating cells for implementing clock-gating functions, specify the desired cells by using the `set_clock_gating_style` command with the following types of options:

```
-gicg_pos_cell  gicg_libcell
-gicg_neg_cell  gicg_libcell
-gicg_pos_auto  gicg_base_libcell
-gicg_neg_auto  gicg_base_libcell
```

Use `-gicg_pos_cell` or `-gicg_neg_cell` (or both) to specify the exact library cell to be used for clock gating. Use `-gicg_pos_auto` or `-gicg_neg_auto` (or both) to have Power Compiler automatically select the best generic integrated clock-gating cell to use, based on the current optimization criteria. For example, with the `-gicg_pos_cell` option, Power Compiler uses the specified library cell to implement clock gating where positive edge-triggered logic is required. The specified cell must be a generic integrated clock-gating cell in the target library.

On the other hand, with the `-gicg_pos_auto` option, Power Compiler uses the specified library cell initially where positive edge-triggered logic is required, but then automatically modifies that choice during optimization if a better generic integrated clock-gating cell can be found in the target library. The library cell you specify with the `-gicg_pos_auto` option is called the “base” cell because it is a simple, basic cell that meets the clock gating requirements. Other cells in the same library might have additional logic or different characteristics. During the compile operation, Power Compiler might substitute a better cell for optimizing the design. It might also modify the logic surrounding the clock-gating cell, allowing the logic functions to be absorbed into the logic of the clock-gating cell, as long as there is a compatible cell in the library.

Examples

Here is a typical command sequence for using a specific generic integrated clock-gating cell (without automatic cell selection):

```
set_clock_gating_style -gicg_pos_cell {my_GICG_lib/
CGLP2a_2en}
read_verilog abc.v
compile_ultra -gate_clock
```

Power Compiler uses the cell `CGLP2a_2en` (in the library `my_GICG_lib`), without modification, to implement clock gating where positive edge-triggered logic is required.

Here is a typical command sequence using generic integrated clock-gating cells with automatic optimized cell selection during the compile operation:

```
set_clock_gating_style -gicg_pos_auto {my_GICG_lib/CGLP1a}
read_verilog abc.v
compile_ultra -gate_clock
```

In this case, Power Compiler initially uses the basic cell CGLP1a to implement clock gating where positive edge-triggered logic is required. This happens during execution of `insert_clock_gating`. During compile, Power Compiler starts at the sequential element of the clock-gating cell, analyzes the surrounding logic, and possibly chooses a better clock-gating cell from the target library. The tool can absorb the logic into the generic clock-gating cell if such a cell is available in the target library.

Library-Level Requirements

Note:

You must specify the `clock_gate_gicg_basic_cell` attribute in the library when your design contains generic integrated clock-gating cells with complex enable conditions.

In Library Compiler, the .lib description of a generic integrated clock-gating cell is different from that of a non-generic cell in the following ways:

- The `clock_gating_integrated_cell` attribute is set to the string `generic` rather than a functional descriptive string.
- The description of the latch in a latch-based integrated clock-gating cell usually has a `statetable` section. A generic cell uses a `latch` description instead.
- The logic definition keyword is `function` rather than `state_function`.

The following examples of Library Compiler .lib syntax demonstrate the differences between non-generic and generic integrated clock-gating cells.

Non-generic integrated clock-gating cell .lib description:

```
cell(CGLP_actlow) {
  area : 1;
  clock_gating_integrated_cell : "latch_posedge";
  dont_use : true;

  statetable(" CP EN ", "IQ ") {
    table : " L  L  : - : L ,\
              L  H  : - : H ,\
              H  -  : - : N ";
  }
  pin(CP_IN) {
    direction : input;
    capacitance : 0.031419;
    clock_gate_clock_pin : true;
    ..
  }
  pin(Z) {
    direction : output;
    state_function : "CP_IN * IQ";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
  }
}
```



```

    timing() {
        timing_sense : positive_unate;
    }

```

Generic integrated clock-gating cell .lib description:

```

cell(CGLP_actlow) {
    area : 1;
    clock_gating_integrated_cell : "generic";
    dont_use : true;

    latch("IQ","IQN") {
        enable : "CP_IN'";
        data_in : "EN * SE'+ SE";
        clear   : "CLR";
    }
    pin(CP_IN) {
        direction : input;
        capacitance : 0.031419;
        clock_gate_clock_pin : true;
        ..
    }
    pin(Z) {
        direction : output;
        function : "CP_IN * IQ";
        max_capacitance : 0.500;
        clock_gate_out_pin : true;
        timing() {
            timing_sense : positive_unate;
        }
    }
}

```

Integrated Clock-Gating Cell Attributes

For non-generic style clock-gating cells, set the `clock_gating_integrated_cell` attribute set to one of 26 function-specific strings, such as `latch_posedge_postcontrol`. Each string is a concatenation of up to four strings that describe the cell's functionality. The library developer specifies the attribute when the integrated cell is created.

The `clock_gating_integrated_cell` attribute can have any one of 26 different values. [Table 7-6](#) contains a short list of example values and their meanings.

Table 7-6 Examples of Values For Integrated Clock Gating Cell

Value of <code>clock_gating_integrated_cell</code>	Integrated cell must contain
<code>latch_negedge</code>	Latch-based gating logic Logic appropriate for gating falling-edge-triggered registers

Table 7-6 Examples of Values For Integrated Clock Gating Cell(Continued)

Value of <code>clock_gating_integrated_cell</code>	Integrated cell must contain
<code>latch_posedge_postcontrol</code>	Latch-based gating logic Logic appropriate for gating rising-edge-triggered registers Test control logic located after the latch
<code>latch_negedge_precontrol</code>	Latch-based gating logic Logic appropriate for gating falling-edge-triggered registers Test control logic located before the latch
<code>none_posedge_control_obs</code>	Latch-free gating logic Logic appropriate for gating rising-edge-triggered registers Test control logic (no latch) Observability port

There are more examples in [Appendix A, “Integrated Clock-Gating Cell Example.”](#)

For a non-generic integrated clock-gating cell, the `set_clock_gating_style` command determines the type of integrated cell that Power Compiler uses for clock gating. Power Compiler searches the library for the integrated cell having the attribute value corresponding to the options you specify with the `set_clock_gating_style` command.

Suppose that you set the clock-gating style as follows:

```
set_clock_gating_style
  -sequential_cell none
  -positive_edge_logic {integrated}
  -control_point before
  -control_signal test_mode]
  -observation_point true
```

When you specify the `-sequential_cell none`, the tool uses a latch-free clock-gating style. In latch-free clock gating you can specify either a `-control_point before` or a `-control_point after`; Power Compiler searches for an integrated clock-gating cell with control as the third string parameter of the `clock_gating_integrated_cell` attribute.

If you use the `-positive_edge_logic` or the `-negative_edge_logic` option of the `set_clock_gating_style` command with the gate type as integrated, the tool inserts an integrated clock-gating cell that is a positive edge latch.

If more than one integrated cell has the correct attribute value, Power Compiler chooses the first integrated cell that it finds in the target library. If you have a preference, be sure to specify the integrated cell by name to ensure that you get the one you want. To fix design rule violations, Power Compiler can size-up the integrated clock-gating cell with another integrated clock-gating cell that is logically equivalent to the one that is being replaced.

Power Compiler does not check the function of the integrated cell to ensure that it complies with the value of the `clock_gating_integrated_cell` attribute. The correct functionality should have been checked by Library Compiler when the integrated cell was initially created. Power Compiler merely searches for an integrated clock-gating cell that contains the attribute value(s) you request.

Pin Attributes

Power Compiler requires certain Library Compiler attributes on the pins of your integrated clock-gating cell. [Table 7-7](#) lists the required pin attributes for pin names that pertain to clock gating. Some pins, such as the pins for test and observability are optional; however, if a pin is present, it must have the corresponding attribute listed in [Table 7-7](#).

Table 7-7 Pin Attributes for Integrated Clock-Gating Cells

Integrated Cell Pin Name	Input or output	Required Library Compiler Attribute
clock	Input	clock_gate_clock_pin
enable	Input	clock_gate_enable_pin
test_mode or scan_enable	Input	clock_gate_test_pin
enable_clock	Output	clock_gate_out_pin
observability	Output	clock_gate_obs_pin

Other tools used in your synthesis and verification flow might require additional pin attributes that are not specific to clock gating and are not listed in [Table 7-7](#).

For more information about Library Compiler attributes and library syntax, see the Library Compiler documentation.

Timing Considerations

Clock gating requires certain timing arcs on your integrated clock-gating cell.

For latch-based clock gating,

- Define setup and hold arcs on the enable pin with respect to the clock pin.
For the latch-based gating style, these arcs are defined with respect to the controlling edge of the clock that is driving the latch.
- Define combinational arcs from the clock and enable inputs to the output.

For latch-free clock gating,

- Define no-change arcs on the enable pin with respect to the clock pin.
For the integrated latch-free gating style, these arcs must be no-change arcs, because they are defined with respect to different clock edges.
- Define combinational arcs from the clock and enable inputs to the output.

For more detailed information about timing your integrated cell, see the Library Compiler documentation.

Propagating Clock Constraints

After creating clock gates, propagate the constraints before compiling your design. The `propagate_constraints` command traverses the hierarchy of the current design, searching for setup- and hold-time constraints on clock-gate subdesigns. The command propagates the setup- and hold-time constraints from the newly created clock-gate subdesigns upward in the design hierarchy. For more information, see the man page for the command.

Ensuring Accuracy When Using Ideal Clocks

When using ideal clocks, set the clock transition time to 0 before analyzing the power of your design. To set the clock transition time to 0, use the `set_clock_transition` command.

The presence of clock-gating circuitry leads to a nonzero transition time on the gated clock signal. This increases with the number of flip-flops being gated by the signal. A large transition time at the clock pin of the gated flip-flop leads to a very high internal power usage. However, this is not realistic because the clock tree synthesis tool inserts buffers to reduce clock edge transition time. Setting the clock transition to 0 ensures the most accurate analysis of timing and power after insertion of clock-gating circuitry and before clock tree synthesis.

Sample Clock-Gating Script

[Example 7-1](#) is a sample script to perform clock gating:

Example 7-1 Clock-Gating Script

```
set_clock_gating_style -sequential latch -min 4 -control_point before \
-control_signal scan_enable -max_fanout 4 -num_stages 6

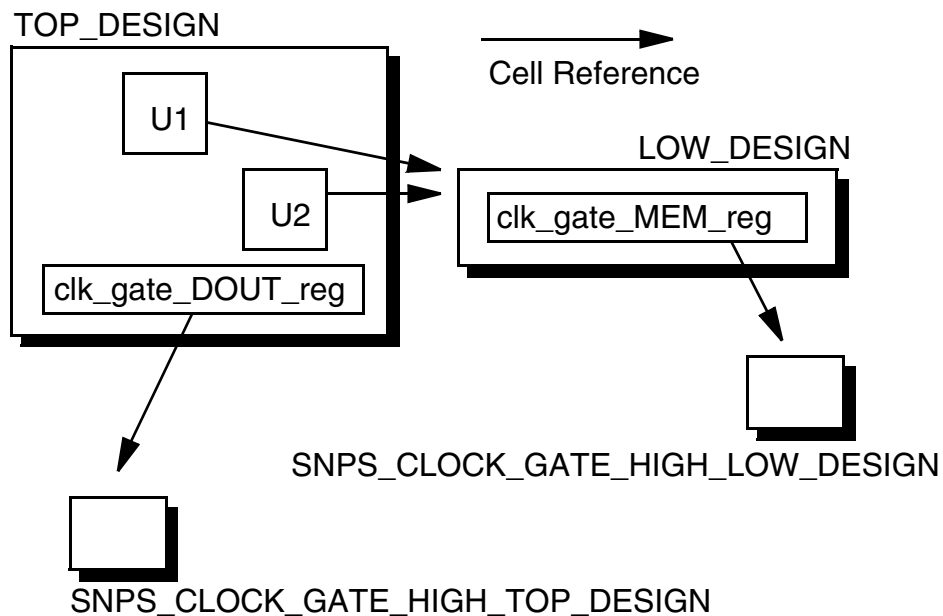
/* analysis and elaboration for clock gating */
analyze -f verilog my_design.v
elaborate TOP_DESIGN # Your top design
current_design TOP_DESIGN

/* clock and constraints */
create_clock clk -p 10
set_fix_hold find(clock, "clk")
set_input_delay 0 -clock clk { reset }
set_input_delay 0 -clock clk { data_in }
set_clock_transition 0 clk

compile_ultra -gate_clock
report_constraints -all_violation
report_clock_gating
report_power
```

The script creates a design having the hierarchy shown in [Figure 7-11](#).

Figure 7-11 Hierarchy of Design With Gated Clocks

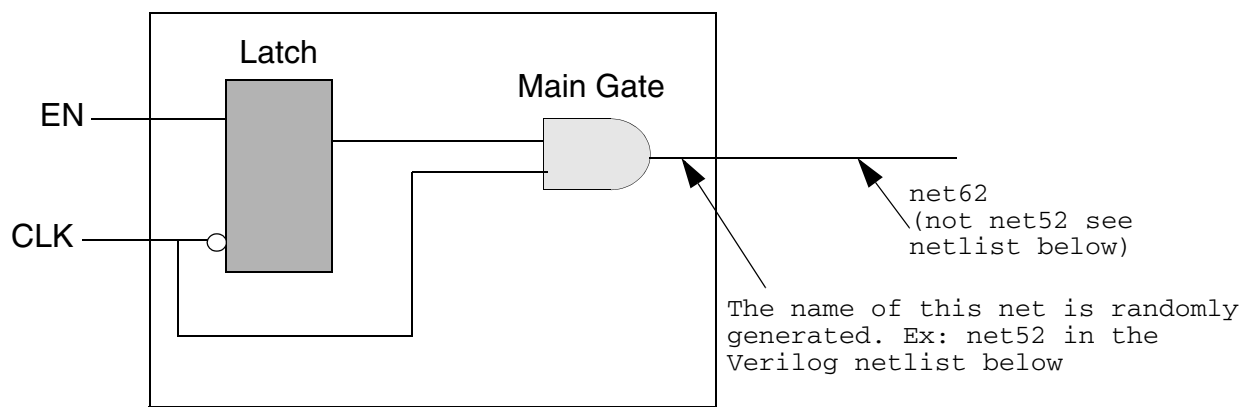


Clock gating creates subdesigns containing clock-gating logic and instantiates them in TOP_DESIGN and LOW_DESIGN.

Clock-Gating Naming Conventions

Clock-gating creates subdesigns containing clock-gating logic as mentioned earlier. Default naming conventions are shown in [Figure 7-12](#).

Figure 7-12 Default Naming Conventions



Module Name: SNPS_CLOCK_GATE_HIGH_<design_name>
Reference Cell Name: clk_gate_<register>

The Verilog netlist may look as follows:

```
module SNPS_CLOCK_GATE_HIGH_ff_03 ( CLK, EN, ENCLK );
    input CLK, EN;
    wire net50, net52, net53, net56;
    assign net50 = CLK;
    assign net50 = CLK;
    assign ENCLK = net52;
    assign net53 = EN;

    L_CSLDP1NQW latch ( .D(net53), .ENN(net50),
    .Q(net56) );
    L_CSAN2 main_gate ( .A(net56), .B(net50), .Z(net52) );
endmodule
module ff_03 ( q, d, clk, e, clr );
    output [2:0] q;
    output [2:0] q;
    input [2:0] d;
    input clk, e, clr;
    wire N0, net62;
```

```

    L_CSFD2QP \q_reg[2] ( .D(d[2]), .CP(net62), .RN(clr),
.Q(q[2]) );
    L_CSFD2QP \q_reg[1] ( .D(d[1]), .CP(net62), .RN(clr),
.Q(q[1]) );
    L_CSFD2QP \q_reg[0] ( .D(d[0]), .CP(net62), .RN(clr),
.Q(q[0]) );
    SNPS_CLOCK_GATE_HIGH_ff_03 clk_gate_q_reg ( .CLK(clk),
.EN(N0),
.ENCLK(net62) );
    L_CSIV1 U5 ( .A(e), .Z(N0) );
endmodule

```

The `module_name(SNPS_CLOCK_GATE_..)`, **reference** `cell_name(clk_gate..)` and the `gated_clock enable net name(net62)` could be changed according to your preferences.

Set the `power_cg_module_naming_style`, `power_cg_cell_naming_style`, and `power_cg_gated_clock_net_naming_style` variables before issuing `insert_clock_gating` command.

Use the variables either in `.synopsys_setup.dc` file or before clock gate insertion. The details of the implementation are as follows:

Usage: set `power_cg_module_naming_style`
`"prefix_%e_%l_midfix_%p_%t_%d_suffix"`
 where,
 prefix/midfix/suffix are just examples of any constant strings that can be specified.
 %e - edge type (HIGH/LOW)
 %l - library name of ICG cell library (if using ICG cells) or concatenated target_library names
 %p - immediate parent module name
 %t - top module (current design) name
 %d - index added if there is a name clash

Usage: set `power_cg_cell_naming_style`
`"prefix_%c_%n_midfix_%r_%R_%d_suffix"`
 where,
 %c - clock
 %n - immediate enable signal name
 %r - first gated reg bank name
 %R - all gated reg banks sorted alphabetically
 %d - index for splitting/name clash resolution

Usage: set `power_cg_gated_clock_net_naming_style`
`"prefix_%c_%e_%g_%d_suffix"`
 %c - original clock
 %e - immediate enable signal name
 %g - clock gate (instance) name

%d - index for splitting/name clash resolution

Sample Script for Naming Style

```
set power_cg_module_naming_style Synopsys_%e_mid_%t
set power_cg_cell_naming_style cg_%c_%n_mid_%R
set power_cg_gated_clock_net_naming_style gclk_%c_%n

define_design_lib WORK -path ./work_writable
set target_library cstarlib_lvt.db
set link_library { cstarlib_lvt.db }

set_clock_gating_style -sequential latch -max_fanout 3 -min 1
analyze -format verilog -lib WORK ff_03.v
elaborate ff_03
insert_clock_gating
uniquify
create_clock -name "clk" -period 5 -waveform { "0" "2.5"
} { "clk" }
compile_ultra
current_design ff_03
write -f verilog -out 3.ff_03.vg -hier
```

Sample Script Output Netlist

```
module Synopsys_HIGH_mid_ff_03_0 ( CLK, EN, ENCLK );
    input CLK;
    input EN;
    output ENCLK;
    wire net15, net12, net11, net9;
    assign net12 = EN;
    assign ENCLK = net11;
    assign net9 = CLK;

    L_CSAN2 main_gate ( .A(net15), .B(net9), .Z(net11) );
    L_CSCLDP1NQW latch ( .D(net12), .ENN(net9), .Q(net15) );
endmodule

module ff_03 ( q, d, clk, e, clr );
    output [2:0] q;
    input [2:0] d;
    input clk;
    input e;
    input clr;
    wire N1, gclk_clk_N1_0;

    Synopsys_HIGH_mid_ff_03_0 cg_clk_N1_mid_q_reg_0 (
.CLK(clk), .EN(N1),
.ENCLK(gclk_clk_N1_0) );
    L_CSFD2QP \q_reg[2] ( .D(d[2]), .CP(gclk_clk_N1_0),
```



```
.RN(c1r), .Q(q[2])
);
  L_CSFD2QP \q_reg[1] ( .D(d[1]), .CP(gclk_clk_N1_0),
.RN(c1r), .Q(q[1])
);
  L_CSFD2QP \q_reg[0] ( .D(d[0]), .CP(gclk_clk_N1_0),
.RN(c1r), .Q(q[0])
);
  L_CSIV1 U3 ( .A(e), .Z(N1) );
endmodule
```

Note:

If there is no %d specified, then Power Compiler assumes a %d at the end.

Keeping Clock-Gating Information in a Structural Netlist

Power Compiler applies several clock-gating attributes to the design and to the clock-gating cells and gated registers in the design. Commands such as `report_clock_gating`, `rewire_clock_gating`, `remove_clock_gating` and several placement optimization algorithms depend on these attributes for proper operation.

The `power_cg_flatten` variable specifies whether to flatten the clock-gating cells when you use commands that perform ungrouping, such as `ungroup`, `compile -ungroup_all`, or `balance_registers`. By default, the variable is set to `false` and the clock-gating cells are not flattened. This is recommended for most situations because ungrouping the clock gates could cause problems.

For example, ungrouped clock gates cannot be mapped to integrated clock gating cells. Power Compiler commands, such as `report_clock_gating`, `remove_clock_gating`, and `rewire_clock_gating`, require the original clock-gating hierarchy. Flattened clock gates are supported when you use integrated clock-gating cells, as long as the flattening is done only after executing the `compile` command.

You can write a structural netlist in ASCII format after clock-gate insertion, synthesis, or placement. Reading back this structural netlist causes the clock-gating attributes to be lost, possibly preventing clock-gating and optimization from operating properly.

If you have used the `compile_ultra -gate_clock` command to insert clock-gating cells, the tool can automatically retrieve the clock-gating attributes and identify the clock-gating cells when you read back the ASCII netlist. For more details see [“Automatic Identification of Clock-Gating Cells” on page 7-58](#)

If you have used the `insert_clock_gating` command to insert the clock-gates, when you save your design in the ASCII format, you must also save the clock-gating attributes using the `write_script` command. Another alternative is to explicitly identify the clock-gating cells using the `identify_clock_gating` command after you read back the design.

Automatic Identification of Clock-Gating Cells

If you insert clock gating in your design, using the `compile_ultra -gate_clock` command, and save the design in ASCII format, the clock-gating attributes are not available when you read back the design in Power Compiler. However, if you set the `power_cg_auto_identify` variable to `true` before you read back the design, Power Compiler can automatically identify the clock-gating cells and the related attributes. The `report_clock_gating` command reports these identified clock-gating cells. Similarly, the `all_clock_gates`, `rewire_clock_gating`, and `remove_clock_gating` commands also identify the clock-gates. [Figure 7-8](#) lists the commands that trigger auto identification of clock gates.

Table 7-8 *Commands that Trigger Identification of Clock Gates*

<code>read_verilog</code>	<code>create_clock</code>	<code>connect_net</code>
<code>read_vhdl</code>	<code>create_design</code>	<code>connect_pin</code>
<code>read_ddc</code>	<code>create_generated_clock</code>	<code>disconnect_net</code>
<code>read_file</code>	<code>elaborate</code>	<code>remove_cell</code>

Note:

Only those clock-gating cells that were inserted by the tool are identified. Clock-gating cells that you manually inserted are not identified by the tool.

Explicit Identification of Clock-Gating Cells

This section discusses the explicit methods of retaining the clock-gating cells and their usage flow.

If you use the `insert_clock_gating` command to insert clock-gating cells when you save the design in ASCII format, you should perform the following to retain the clock-gating information.

- Save the attributes settings using the `write_script` command.
- Use the `identify_clock_gating` command after you read back the ASCII netlist.
This command regenerates the clock-gating attribute settings.

The advantages and disadvantages of these two methods are summarized in [Table 7-9](#).

Table 7-9 Identifying Clock-Gated Designs

Strategy	Advantages	Disadvantages
<code>write_script</code>	Clock-gating attributes are written out in <code>set_attribute</code> commands to save current settings. Familiar command and procedure.	Netlist changes may not be supported
<code>identify_clock_gating</code>	Netlist changes performed outside of Design Compiler are supported.	Invoke this command at the right place. Some attributes such as <code>max_fanout</code> might be lost unless the <code>set_clock_gating_style</code> command is used.

Usage Flow With the `write_script` Command

Follow these steps to retrieve the clock-gating information in the ASCII netlist, using the `write_script` command.

1. Setup environment; read in the RTL design and insert the clock-gating logic.
2. Compile the design with the required constraints.
3. Run the `change_names` command to conform to the specified rules.
4. Write out the netlist
5. Save current attributes and settings by using `write_script -hier` command. Use the `-o` option of the command to write the output to a file. This command writes out all the attributes set by the `set_attribute` command.
6. Quit the Design Compiler session. Make sure you do not make any changes to the netlist before quitting.
7. Read in the design netlist.
8. Source the file written by the `write_script` command. This sets all the required attributes on the design, including the clock-gating cells, for proper execution throughout the flow.

If you do not need the clock-gating information, you can use the `-no_cg` option of the `write_script` command. This results in a smaller script file.

You can now report the identified clock gates, using the `report_clock_gating` command.

The following example script shows the output file created by the `write_script` command.

```
#####

# Created by write_script -format dctl on August 26, 2009 8:09 am

#####

# Set the current_design #
current_design module4

set_local_link_library {CORELIB8DLL.db}
set_attribute -type int [current_design] power_cg_max_fanout
2048
set_attribute -type boolean [get_cells clk_gate_out1_reg] \
clock_gating_logic true
set_attribute -type boolean [get_cells clk_gate_out1_reg] \
hpower_inv_cg_cell false
set_attribute -type integer [get_cells {out1_reg[0]}] \
power_cg_gating_group 0
set_attribute -type integer [get_cells {out1_reg[1]}] \
power_cg_gating_group 0
set_attribute -type integer [get_cells {out1_reg[2]}] \
power_cg_gating_group 0
set_attribute -type integer [get_cells {out1_reg[3]}] \
power_cg_gating_group 0
set_attribute -type integer [get_cells {out1_reg[4]}] \
power_cg_gating_group 0
set_attribute -type integer [get_cells {out1_reg[5]}] \
power_cg_gating_group 0
set_attribute -type integer [get_cells clk_gate_out1_reg] \
power_cg_gating_group 0
set_size_only [get_cells latch] true
```

Usage Flow With the `identify_clock_gating` Command

This section describes the steps you follow to retrieve the clock-gating information, using the `identify_clock_gating` command.

After you have saved the design that has the clock-gating information, follow these steps to retrieve the clock-gating information:

1. Read in the manipulated structural netlist that already has clock-gating cells inserted.
2. Set the constraints, at the least the clock constraint that was used earlier. This ensures the number of clocks in the designs that was used for clock-gating optimization.
3. Set the `set_clock_gating_style` command. This ensures that the settings are the same as before saving the design. Otherwise, a few attributes such as `max_fanout` are not retained.

4. Use the `identify_clock_gating` command without any options to identify all clock-gating elements. This step ensures that the design is traversed and searched appropriately for the clock-gating structure that is inserted by Power Compiler and annotates the attributes needed for later operations. When you do not specify any option, the `identify_clock_gating` command traverses only those clocks that were specified using the `create_clock` command.

Note:

The `create_clock` command is not necessary when options are used with the `identify_clock_gating` command.

Your design now contains all the clock-gating information. You can verify this using the `report_clock_gating` command.

[Table 7-10](#) summarizes the options of the `identify_clock_gating` command.

Table 7-10 Options for the `identify_clock_gating` Command

Argument	Description
<code>-reset</code>	Resets all clock-gating attributes.
<code>-reset_only</code>	Cell or netlist objects need to be specified for this option. Resets the clock-gating attributes on these objects only.
<code>-gating_elements</code>	Marks the specified cell as a gating element. Could be used to fix any problems.
<code>-gated_elements</code>	Marks a cell or a pin that is specified as a gated element. Could be used to fix any problems.
<code>-ungated_element</code>	Marks specified cells as cells that have no clock gating. Could be used to fix any problems by previous command.

Replacing Clock-Gating Cells

Power Compiler is capable of automatically detecting gating circuitry at the block or module level. The gating circuit can be either hand instantiated or inferred logic at the module level. Power Compiler replaces this logic with an integrated clock-gating cell or discrete cells as per the user specifications in the `set_clock_gating_style` command. This operation is performed using the `replace_clock_gates` command. This feature allows you to use the integrated clock-gating circuit. Additionally, Power Compiler commands, such as the `report_clock_gating`, `remove_clock_gating`, and `rewire_clock_gating`, recognize this cell for further operations.

Power Compiler needs to know the edge of the registers that the module clock gates are driving. If the clock gate drives macro cells or black box cells for which the edge cannot be determined, then the module replacement is not performed. To define the edge type for the cells in the fanout of the gating cell, use the `replace_clock_gates` command.

Follow these steps to perform module-level replacement of clock-gating cells:

1. Set clock-gating directives and styles (optional)

The default values of the `set_clock_gating_style` command is suitable for most designs. You can choose a value for the clock-gating conditions, and a clock-gating style that is compatible with the clock-gating cell that is being replaced using the `set_clock_gating_style` command.

2. Read the RTL design.

3. Define the clock ports.

The clock port must be identified using the `create_clock` command before performing replacement operation.

4. Insert clock-gating cells.

Use the `insert_clock_gating` or the `compile_ultra -gate_clock` commands. Power Compiler inserts clock gates according to the default values of the clock-gating style if a style is not specified.

5. Replace manually instantiated clock-gating cells.

Use the `replace_clock_gates` command. Power Compiler replaces manually inserted clock gates with the tool inserted clock gates according to the default values of the style if a style is not specified earlier. Use the `-global` option to perform the replacement hierarchically.

6. Report the gate elements registers.

Use the `report_clock_gating` command to get the list of cells as shown in the following example:

```
dc_shell> read_verilog design.v
dc_shell> create_clock -period 10 -name clk
dc_shell> compile_ultra -gate_clock
dc_shell> replace_clock_gates -global
dc_shell> report_clock_gating
dc_shell> report_power
```

In the following example, replacement is performed on a gating cell that is driving registers in a black box cell:

```
dc_shell> read_verilog design.v
dc_shell> create_clock -period 10 -name clk
dc_shell> set_replace_clock_gates -rising_edge_clock RAM/clk
```

```
dc_shell> compile_ultra -gate_clock
dc_shell> replace_clock_gates -global
dc_shell> report_clock_gating
```

In the following example, replacement is performed only on selected gating cells:

```
dc_shell> read_verilog design.v
dc_shell> create_clock -period 10 -name clk
dc_shell> set_replace_clock_gates -exclude_instances {SUB/C10}
dc_shell> compile_ultra -gate_clock
dc_shell> report_clock_gating
```

[Example 7-2](#) shows a clock-gate replacement report.

Example 7-2 Clock-Gate Replacement Report

```
Current clock gating style....
Sequential cell: none
Minimum register bank size: 3
Minimum bank size for enhanced clock gating: 6
Maximum fanout: 2048
Setup time for clock gate: 1.300000
Hold time for clock gate: 0.000000
Clock gating circuitry (positive edge): or
Clock gating circuitry (negative edge): and
Note: inverter between clock gating circuitry
      and (negative edge) register clock pin.
Control point insertion: none
Control signal for control point: scan_enable
Observation point insertion: false
Observation logic depth: 5
Maximum number of stages: 5
1
replace_clock_gates -global
Loading target library 'ssc_core_typ'
Loading design 'regs'
Information: Performing clock-gating on design regs
```

Clock Gate Replacement Report

Clock	Cell Name	Include	Clock	Edge	Func.	Setup	Gate
Root		Exclude	Fanin	Type		Cond.	Repl.
clk	C7	-	1	fall	and	yes	yes

Summary:

	number	percentage
Replaced cells (total):	1	100
Cell not replaced because		
Cell was excluded:	0	0
Multiple clock inputs:	0	0
Mixed or unknown clock edge type:	0	0
No compatible clock gate available:	0	0
Setup condition violated:	0	0
Total:	1	100

Clock Gate Insertion Report

Gated Group	Flip-Flop Name	Include Exclude	Bits	Enable Cond.	Setup Cond.	Width Cond.	Clock Gated
cg0	GATED REGISTERS						
	q2_reg[3]	-	1	yes	yes	yes	yes
	q2_reg[2]	-	1				
	q2_reg[1]	-	1				
cg1	q2_reg[0]	-	1				
	q3_reg[3]	-	1	yes	yes	yes	yes
	UNGATED REGISTERS						
	si_reg	-	1	no	??	??	no
	ti_reg	-	1	no	??	??	no
	q4_reg[0]	-	1	no	??	??	no

Summary:

Flip-Flops	Banks		Bit-Width	
	number	percentage	number	percentage
Clock gated (total):	3	30	12	54
Clock not gated because				
Bank was excluded:	0	0	0	0
Bank width too small:	0	0	0	0
Enable condition not met:	7	70	10	45
Setup condition violated:	0	0	0	0
Total:	10	100	22	100

Clock gates in design	number		percentage	
Replaced clock gates:	1		16	
Inserted clock gates:	3		50	
Factored clock gates:	2		33	
Total:	6		100	

Multistage clock gating information

Number of multistage clock gates:	2
Average multistage fanout:	2.0
Number of gated cells:	16
Maximum number of clock gate stages:	3
Average number of clock gate stages:	2.2

Optimizing Clock-Gate Insertion

In order to further increase the power saving of your design, Power Compiler uses certain techniques to reduce the number of clock-gating cells in the design. Some of these techniques are multistage clock-gating, hierarchical clock gating. These techniques are described in detail in the following sections:

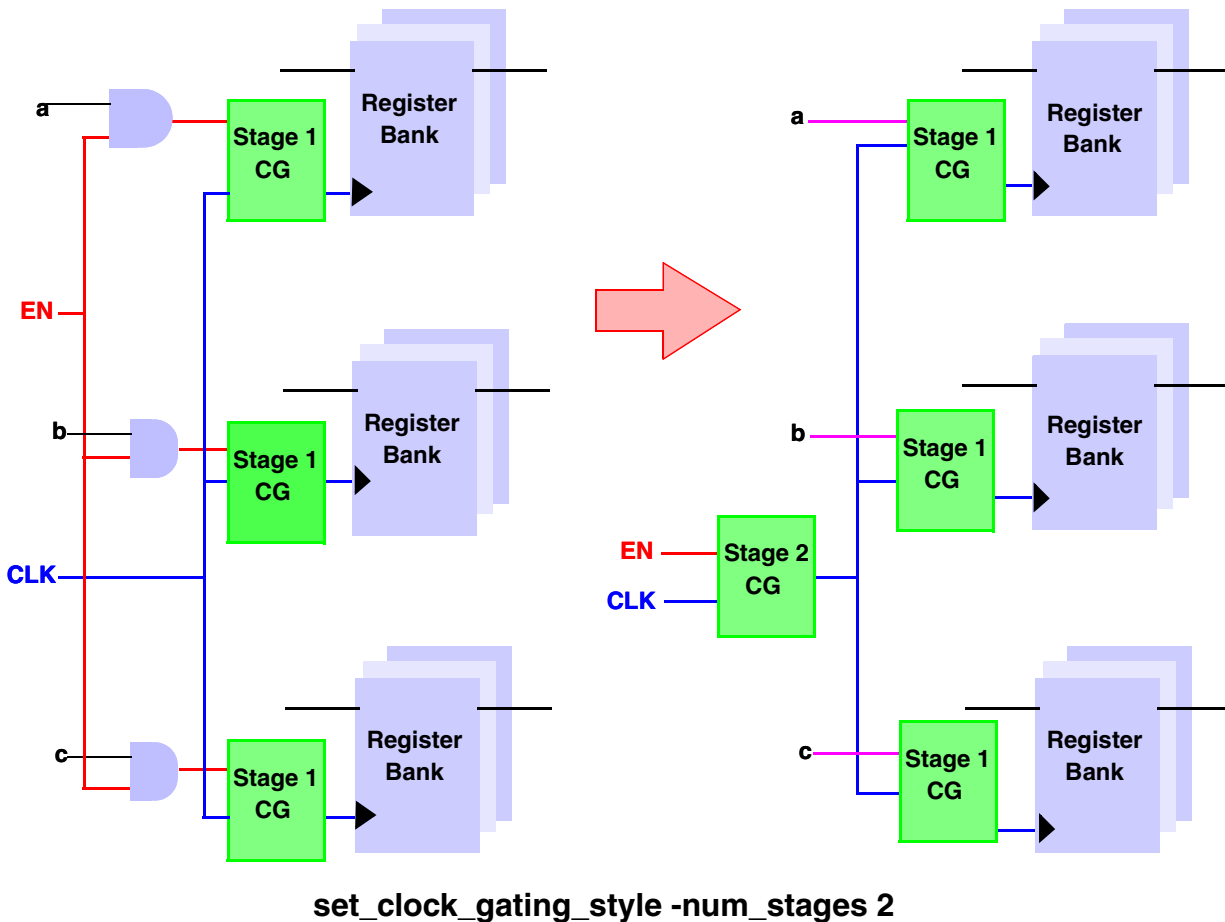
Multistage Clock Gating

When a clock-gating cell is driving another or a row of clock-gating cells, this is referred to as multistage clock gating. Power Compiler can identify common enables and factoring using another clock-gating cell as shown in [Figure 7-13](#). The tool can apply multistage clock gating not only on RTL designs but also on designs that contain gate cells, for further optimizations if available.

The multistage clock-gating feature allows you to combine as many register banks as possible so that the clock gating can be moved up closer to the top, leading to more power savings. As a result, the actual benefits are seen when combined with placement.

For the tool to perform multistage clock gating, you should set the maximum number of stages for multistage clock gating using the `-num_stages` option of the `set_clock_gating_style` command. The default value of the `-num_stages` option is 1. After setting the maximum number of stages, use either `compile_ultra -gate_clock` or `insert_clock_gating` command to perform multistage clock gating.

Figure 7-13 Multistage Clock Gating



However, using the `compile` or `compile_ultra` command you can perform the following additional tasks during multistage clock gating.

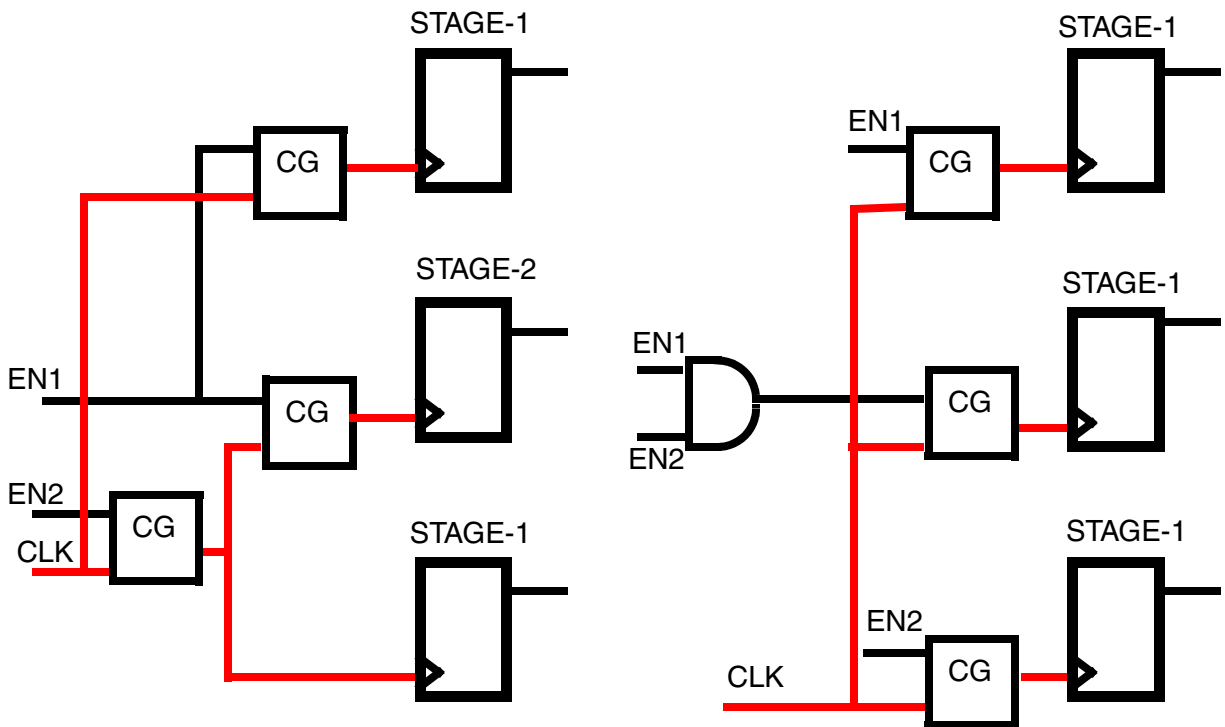
- Reconfigure the number of clock-gating stages

If you set the `power_cg_reconfig_stages` variable to `true`, the tool reconfigures the number of clock-gating stages. The reconfiguration complies with the value of the `-num_stages` option of the `set_clock_gating_style` command. This is done only on the clock gates inserted by the tool and the integrated clock-gating cells (ICG).

- Balance the number of clock-gating stages

If you set the `power_cg_balance_stages` variable to `true`, the tool balances the number of clock-gating stages across various register banks. Balanced clock-gate stages ensure uniform clock latency across register banks. Figure 7-14 shows the transformation for balancing the clock-gating stages.

Figure 7-14 Balancing the number of stages



Multistage Clock Gating Flow

Follow the steps mentioned below to perform multistage clock gating on your design:

1. Set clock-gating styles and directives.

Use the `set_clock_gating_style` command to specify the clock gating stages and other clock gating conditions. The clock gating options you set should be compatible with the functionality of the clock-gating cell that is being replaced and of the registers. You can set the number of stages for multistage clock gating as shown below:.

```
set_clock_gating_style -num_stages 5
```

The default for the `-num_stages` option is 1. This implies that when the `-num_stages` option is not used, further factoring is not done by the tool.

2. Read your design.

Read in the design using a read command.

3. Multistage clock gating.

Use the `compile_ultra -gate_clock` or `insert_clock_gating` command.

4. Report the gate elements registers.

Use the `report_clock_gating` command to get the list of cells and the `report_power` command to see the design power after the multistage clock gating.

The following is a sample script to perform multistage clock gating using the `compile_ultra -gate_clock` command:

```
# set the target library and the link library

set_clock_gating_style -num_stages 2
read_verilog design.v
create_clock -name clk -period 10
compile_ultra -gate_clock
report_clock_gating -verbose -gating -gated
report_power
```

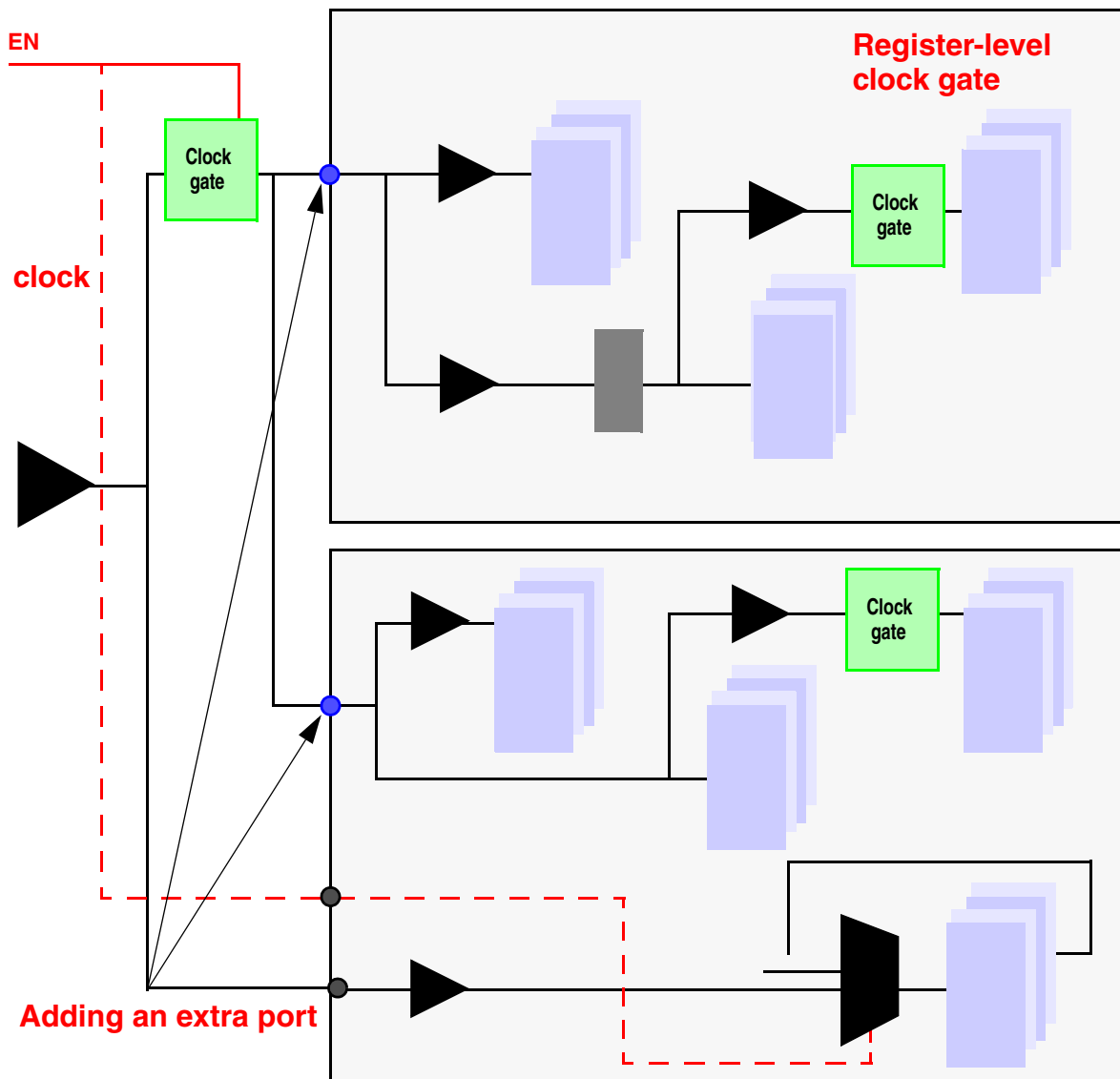
Hierarchical Clock Gating

Generally clock gating techniques in Power Compiler extracts common enable conditions that are shared across the registers within the same block.

In hierarchical clock gating, Power Compiler extracts the common enables shared across registers in different levels of hierarchy in the design, during the clock-gate insertion. This technique looks for globally shared enables while inserting clock gating cells. This increases the clock-gating opportunity and also reduces the number of clock-gate insertion. With this technique and proper placement, more power savings can be obtained.

Power Compiler inserts hierarchical clock-gating cells at various levels of design hierarchy. As a result, additional ports are created for the clock-gated enable signal as shown in [Figure 7-15](#). These additional ports are added to the subdesigns. Formality verifies the designs successfully as long as the designs being compared have the same number of primary ports.

Figure 7-15 Additional Ports During Hierarchical Gating



Power Compiler can perform hierarchical clock gating on RTL netlists as well as gate-level netlists. Use the `compile_ultra -gate_clock` or the `insert_clock_gating -global` command to perform hierarchical clock gating. You use the `compile_ultra -gate_clock` command to perform hierarchical clock-gating on both RTL and gate-level netlists. This command is especially useful for clock gating on gate-level netlists.

To perform hierarchical clock gating using the `compile_ultra -gate_clock` command, you must set the `compile_clock_gating_through_hierarchy` variable to `true` before compiling your design. If you use the `insert_clock_gating` command, you must use the `-global` option.

Steps involved in the hierarchical clock gating flow:

1. Read the design
2. Set clock-gating styles and directives (optional)

The default values of the `set_clock_gating_style` command are suitable for most designs. You can use this command to choose a style.

3. Set the `compile_clock_gating_through_hierarchy` variable to `true` and compile your design using the `compile_ultra -gate_clock` command.

Alternatively, you can use the `insert_clock_gating -global` command before compiling your design. The `-global` option not only inserts clock gating globally if it finds the commonly shared enables across subdesign blocks, but it also performs the general clock-gate insertion for registers that have unique enable signals.

Note:

Without the `-global` option, the `insert_clock_gating` command processes each design separately, independent of its hierarchy, and performs limited constant propagation.

The following sample script demonstrates hierarchical clock gating using the `compile_ultra` command.

```
# Set your target library and link library
# set_clock_gating_style (optional)
# Following command is optional. Use for global clock gating
set compile_clock_gating_through_hierarchy true
# Read your design
create_clock -name clk -period 10
compile_ultra -gate_clock
report_clock_gating -ver -gating -gated -multi_stage
report_power
```

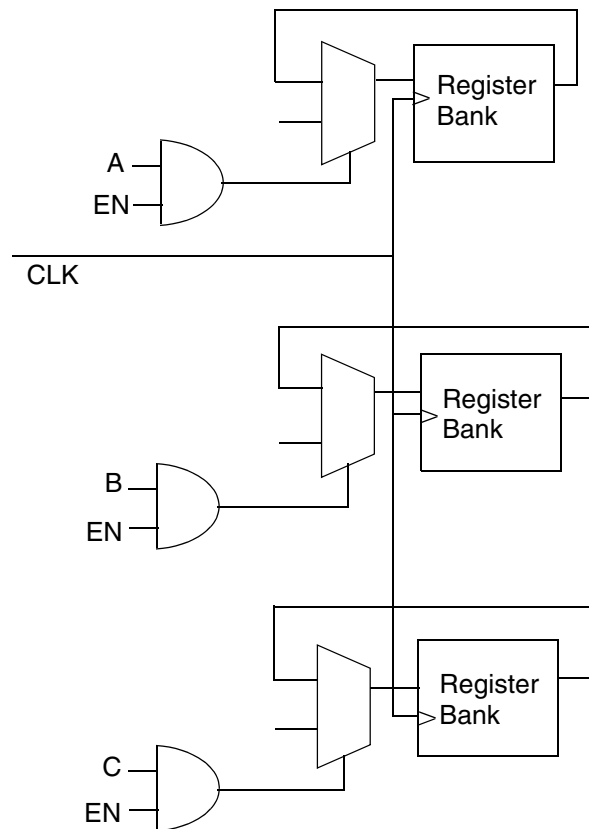
Enhanced Register-Based Clock Gating

The regular register-based clock gating requires certain conditions in order for successful implementation. One of these conditions is the minimum bit width of the register bank to be gated. If the minimum bit width is less than 3, which is the default, there is no clock-gating opportunity. This width constraint ensures that the overhead of using the clock-gating cell does not overcome the power savings.

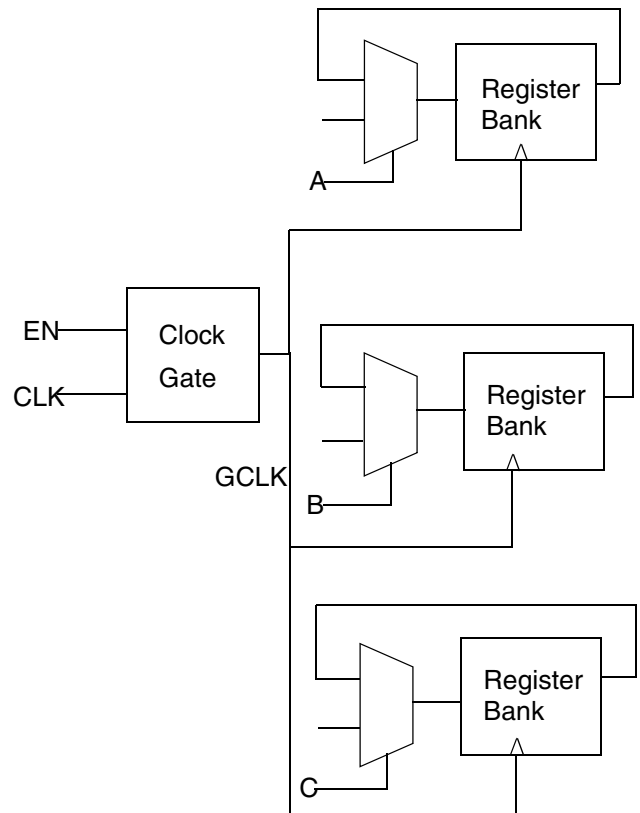
Power Compiler can factor out the common enable signal EN shared between three register banks and insert one clock-gating cell for these register banks, which would normally not be clock gated due to the width condition. The result is shown in [Figure 7-16](#).

Figure 7-16 Design With Common Enable Signal

**Width Condition Violation ($W=2$):
No Clock Gating**



Common Enable Factoring:



The default total minimum bit width of registers for enhanced clock gating to be implemented is twice that of regular clock gating. Since the default for regular register clock gating is 3, for the enhanced clock gating the register width should be at least $2 * 3$, which is 6.

Enhanced clock gating is done by default along with regular clock gating with the `insert_clock_gating` command. To turn off enhanced clock gating use the `-regular_only` option of the `insert_clock_gating` command.

In the following example, automated clock gating, along with enhanced clock gating, is implemented if the conditions are met.

```
dc_shell> read_verilog design.v
dc_shell> create_clock -period 10 -name clk
dc_shell> insert_clock_gating
dc_shell> report_clock_gating
```

In the following example, enhanced clock gating is turned off:

```
dc_shell> read_verilog design.v
dc_shell> create_clock -period 10 -name clk
dc_shell> insert_clock_gating -regular_only
dc_shell> report_clock_gating
```

Performing Clock-Gating on DesignWare Components

Power Compiler provides the ability to perform clock gating on DesignWare components instead of treating them as black box cells. The `compile_ultra -gate_clock` command perform clock gating on DesignWare components, by default.

If you use the `insert_clock_gating` command to insert clock gates, to run clock gating on designware components set the `power_cg_designware` variable to true. The default value of this variable is false.

Shown below is a sample script to perform clock gating on DesignWare components:

```
set power_cg_designware true
set target_library [list my_lib.db cg_integ_pos.db]
set synthetic_library dw_foundation.sldb
set link_library [list "*" my_lib.db
dw_foundation.sldb cg_integ_pos.db]
set_clock_gating_style -min 1 -sequential_cell latch -pos \
    {integrated:CGLP} # Optional
read_verilog cpurd_fifo.v
write -verilog -hier -o elab.v
compile_ultra -gate_clock
insert_dft
write -verilog -hier -o comp.v
```

You can view the DesignWare clock-gated registers using the `report_clock_gating -gated` command. The DesignWare clock gates are designated with (*) in the report.

Reporting Command For Clock Gates and Clock Tree Power

The `report_clock_gating` command reports the clock-gating cells, the gated and the ungated registers in the current design. To see the dynamic power savings because of clock gate insertion, use the `report_power` command before and after the clock-gate insertion.

The top portion of the report indicates the name of each register, the clock-gating conditions the flip-flop satisfies, and whether or not the flip-flop's clock was gated. The double question mark (??) indicates that the condition was not checked during clock gating because a previously checked clock-gating condition was not satisfied. All conditions must be satisfied to gate the clock, unless you use the `set_clock_gating_registers` command.

The Gated Group column contains arbitrary names for groups of register banks that have equivalent enable signals. Power Compiler creates the group names during clock gating and uses one clock gate to gate the register banks in each group.

In the summary portion of the report, the Banks columns show the total number and percentages of flip-flops with gated and ungated clocks. In the Bit-Width columns, the report shows cumulative bits and percentages of total bits for gated and ungated flip-flops.

The `report_clock_gating` Command

The following samples are the output of the `report_clock_gating` command. If you use the `report_clock_gating` command without any option, the summary of the clock-gating elements in the current design is printed as shown in [Example 7-3](#).

Example 7-3 Clock-Gating Report Using Default Settings

```
dc_shell> report_clock_gating
*****
Report : clock_gating
Design : low_design
Version: X-2005.09
Date   : August 26, 2009 8:09 am
*****
```

Clock-Gating Summary

Number of Clock gating elements	1
Number of Gated registers	4 (66.67%)
Number of Ungated registers	2 (33.33%)
Total number of registers	6

[Example 7-4](#) shows a sample report using the `-gating_elements` option.

Example 7-4 Clock-Gating Report Using the `-gating_elements` Option

```
dc_shell> report_clock_gating -gating_elements
*****
Report : clock_gating
        -gating_elements
Design : low_design
Version: X-2005.09
```

```

Date    : August 26, 2009 8:09 am
*****

-----
                        Clock-Gating Cell Report
-----

Clock Gating Bank : clk_gate_out1_reg (ss_hvt_0v70_125c: 0.7)
-----

STYLE = latch, MIN = 2, MAX = unlimited, HOLD = 0.00, OBS_DEPTH = 5

INPUTS :
    clk_gate_out1_reg/CLK = clk
    clk_gate_out1_reg/EN = N6
    clk_gate_out1_reg/TE = test_se

OUTPUTS :
    clk_gate_out1_reg/ENCLK = net107

Clock Gating Bank : sub/clk_gate_out_reg (ss_hvt_1v08_125c: 1.08)
-----

STYLE = latch, MIN = 2, MAX = unlimited, HOLD = 0.00,
OBS_DEPTH = 5

INPUTS :
    sub/clk_gate_out_reg/CLK = n22
    sub/clk_gate_out_reg/EN = N6
    sub/clk_gate_out_reg/TE = test_se

OUTPUTS :
    sub/clk_gate_out_reg/ENCLK = net95
-----

```

Example 7-5 shows a sample report using the `-ungated`, `-gated`, `-gating_elements`, and `-verbose` options. A table is created to display all the ungated and gated registers in your current design.

Example 7-5 Clock-Gating Report using Gated and Ungated Elements

```

*****
Report : clock_gating
        -gating_elements
        -gated
        -ungated
        -verbose
Design : regs
Version: A-2007.12
Date    : August 26, 2009 8:09 am
*****
-----
                        Clock Gating Cell Report
-----

```

```
-----
Clock Gating Bank : clk_gate_C7
-----
```

```

STYLE = none, MIN = 3, MAX = 2048, HOLD = 0.00, SETUP = 1.30,
OBS_DEPTH = 5
TEST INFORMATION :
OBS_POINT = NO, CTRL_SIGNAL = scan_enable, CTRL_POINT = none
INPUTS :
clk_gate_C7/CLK = clk
clk_gate_C7/EN = xi
OUTPUTS :
clk_gate_C7/ENCLK = xclk
RELATED REGISTERS :
q4_reg[3]
q4_reg[2]
q4_reg[1]
q4_reg[0]
-----
```

Gated Register Report

Clock Gating Bank	Gated Register
clk_gate_C7	q4_reg[0] q4_reg[1] q4_reg[2] q4_reg[3]
clk_gate_q3_reg	q3_reg[0] q3_reg[1] q3_reg[2] q3_reg[3]

Ungated Register Report

Ungated Register	enable	ctrl	width	removed
xi_reg	NO	-	-	-
ti_reg	NO	-	-	-
si_reg	NO	-	-	-

Clock Gating Summary

Number of Clock gating elements	6
Number of Gated registers	16 (72.73%)
Number of Ungated registers	6 (27.27%)
Total number of registers	22

Example 7-6 shows a report generated with the `-multi_stage` and `-no_hier` options for a hierarchical multistage clock gated design. A multistage clock gate is a clock-gating cell that is driving another clock-gating cell. The report shows three clock-gating elements, eight gated and no ungated registers at the top level. Two of the three clock gates are multistage, and their average fanout is 1.0, indicating that the clock path consists of a chain of three clock gates. There is one gated module in addition to the eight gated registers. The eight registers have three stages on their clock path, but the module has only two, bringing the average number of stages to $2.9 = ((8*3 + 2*1)/9)$.

Example 7-6 Clock-Gating Report Using the `-no_hier` and `-multi_stage` Options

```
*****
Report : clock_gating
        -no_hier
        -multi_stage
Design  : regs
Version: X-2005.09
Date    : August 26, 2009 8:09 am
*****
```

Number of Clock gating elements	6
Number of Gated registers	16 (72.73%)
Number of Ungated registers	6 (27.27%)
Total number of registers	22
Number of multi-stage clock gates	2
Average multi-stage fanout	2.0
Number of gated cells	16
Maximum number of stages	3
Average number of stages	2.2

Example 7-7 Clock-Gating Report Using the `-style` Option

```
-----
                        Clock Gating Style Report
-----

Clock Gating Style 1 : (3 clock gates)
-----

STYLE
    sequential_cell latch
    minimum_bitwidth 2
    enhanced_min_bitwidth 4
```

```

positive_edge_logic integrated:TLATNTSCAX12MTH
negative_edge_logic or
control_point before
control_signal scan_enable
observation_point flase
num_stages 2

```

CLOCK GATES

```

clk_gate_out1_reg
sub/clk_gate_out_reg
sub/r0/clk_gate_out_reg

```

Clock Gating Summary

Number of Clock gating elements	3
Number of Gated registers	16 (100.00%)
Number of Ungated registers	0 (0.00%)
Total number of registers	16

You use the `report_clock_gating -structure` to get the details and a summary of the clock-gating structure.

Note:

You cannot use the `-structure` option along with any other option.

Example 7-8 shows the clock-gating report when you specify the `-structure` option.

Example 7-8 Clock-Gating Report Using the -structure Option

```

*****
Report : clock_gating
        -structure
Design : test
Version: B-2008.09
Date   : Mon Jul 14 15:01:12 2008
*****

```

Clock Gating Structure Summary

Clock	Total Registers	CG Stage	# of Clock Gates	# of Gated Cells
clka	284	1	9	285
clkb	284	1	9	285

Clock Gating Structure Details

Clock	CG Stage	Gating Element	Fanout	Latency	Gated Cells

clk_a	1	cg_1	2	0.000	macro_inst
		clk_gate_y_reg	132	0.000	S4/y_reg[0] S4/y_reg[1] S4/y_reg[2] S4/y_reg[22] S4/y_reg[23]
		S7/clk_gate_y_reg	4	0.000	S7/y_reg[0] S7/y_reg[1] S7/y_reg[2] S7/y_reg[3]
		S8/clk_gate_y_reg	4	0.000	S8/y_reg[0] S8/y_reg[1] S8/y_reg[2] S8/y_reg[3]
		S9/clk_gate_y_reg	4	0.000	S9/y_reg[0] S9/y_reg[3] S9/y_reg[1] S9/y_reg[2]

Clock Gating Summary

Number of Clock gating elements	9
Number of Gated registers	285 (100.00%)
Number of Ungated registers	0 (0.00%)
Maximum number of stages	1
Total number of registers	285

8

Operand Isolation

Operand Isolation is a technique used by Power Compiler to reduce dynamic power consumption for datapath designs.

This chapter includes the following sections:

- [Operand Isolation Overview](#)
- [Operand Isolation Methodology Flows](#)
- [Commands and Variables Related to Operand Isolation](#)
- [Using Operand Isolation](#)
- [Interoperability](#)
- [Debugging Tips](#)
- [Examples](#)

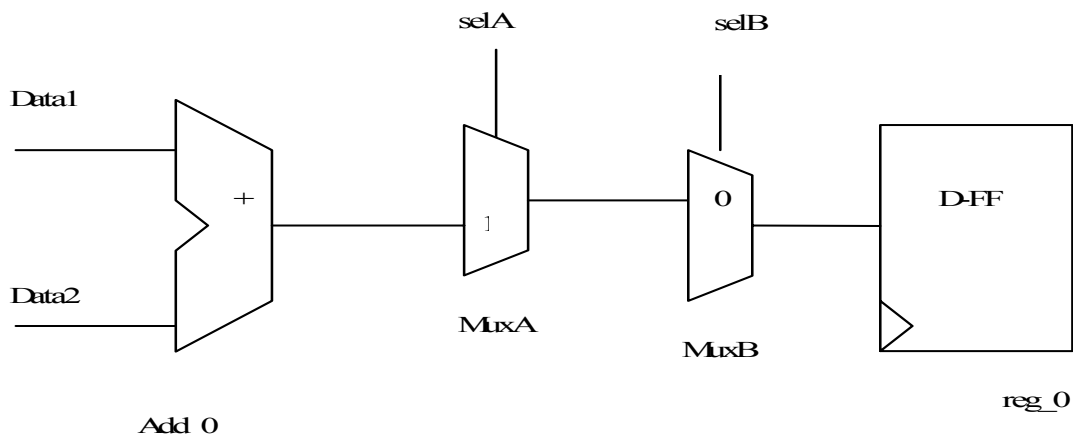
Operand Isolation Overview

In a datapath intensive design, the complex combinational circuits may contribute to the majority of power consumption of the design. If the fan out of a datapath circuit is not observed under conditions, the operand isolation approach can reduce the dynamic power or dissipation of the circuit by adding isolation logic such as AND or OR gates along with the control signal to hold the inputs of the datapath operator to a constant. Therefore, no switching activity at the inputs propagates through the circuit and no redundant computations are performed.

To illustrate this scenario, consider the following example:

In [Figure 8-1](#), the adder (cell name Add_0) consumes power whenever the input Data1 and Data2 toggle. However, the output of the adder is observed at the flip-flop (reg_0) input only when selA is “1” and selB is “0”.

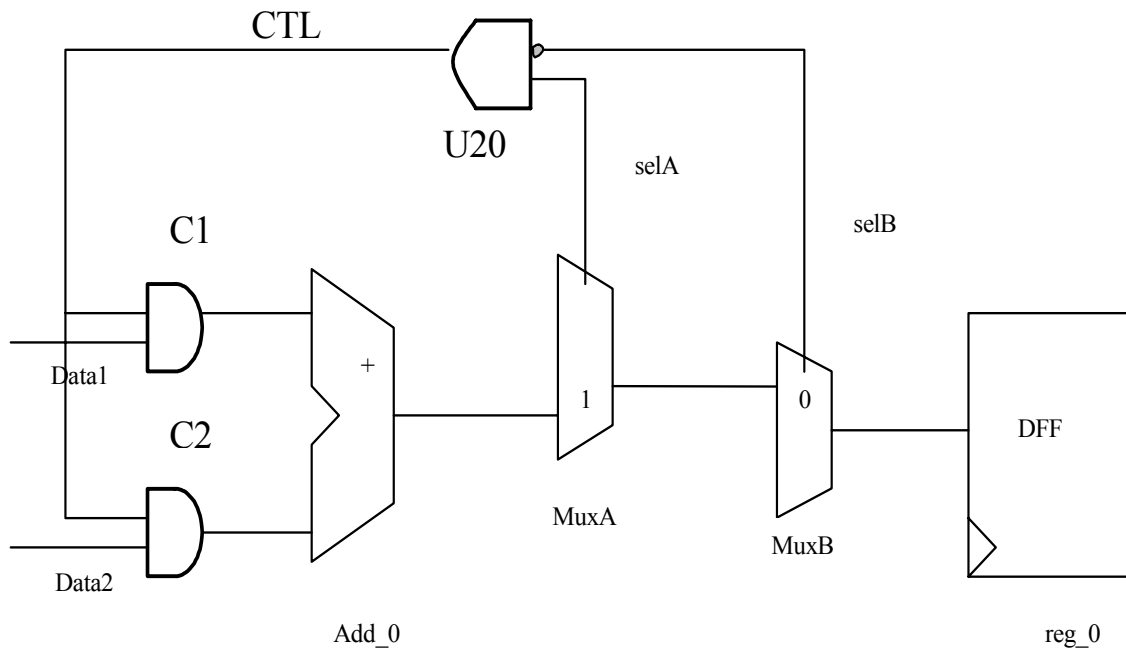
Figure 8-1 Operand Isolation Candidate



[Figure 8-2](#) shows an example of applying operand isolation. Power Compiler inserts isolation logic to gate the inputs of Add_0.

In this example, the isolation logic consists of a control signal CTL and some AND gates. The inputs to the adder are enabled by CTL, while $CTL = selA * !selB$. The inserted gates are C1, C2 and U20.

Figure 8-2 Design With Operand Isolation



Observable Don't Care Conditions

The ideal candidates for operand isolation are combinational circuits with some complexity such as arithmetic logic units (ALUs), wide-bus adders, multipliers and hierarchical combinational cells that frequently perform redundant computation. In order to be able to perform operand isolation, the fan out of the combinational circuit needs to have an observable don't care (ODC) condition. If the output of the circuit is always observed, there's no operand isolation opportunity.

Referring to the example in [Figure 8-1](#), two multiplexers (MuxA and MuxB) generate the ODC conditions.

For more information, see [“Verilog RTL With Observable Don't Care Conditions” on page 8-22](#).

Power Compiler Operand Isolation Approach

Power Compiler performs power-based, automatic, RTL-level exploration of operand isolation. In general, Power Compiler applies operand isolation on an object if all of the following conditions are met:

- The object is an arithmetic operator or a combinational hierarchical cell.

- The fan out of the object has ODC conditions.
- The netlist exploration process indicates that by inserting operand isolation it will most likely to reduce the dynamic power consumption of the circuit.

Several factors can influence operand isolation:

- The static probability and toggle rate; for example, the switching activity at the input data net of the operator.
- The SP and toggle rate of the nets which generate the control signal.
- The complexity of the isolation objects.

For the example in [Figure 8-2](#), assume selA and selB are uncorrelated. The probability of selA==1 is 0.9 and the SP probability of selB ==0 is also 0.9. Since the cell (U20) which generates the control signal is an AND gate, the probability for the control signal CTL==1 (the case when the adder output is observed) is $0.9 \times 0.9 = 0.81$. This indicates the adder output would be observed by the flip flop for the majority of the time. By default, Power Compiler does not perform operand isolation on the adder since the reduction of total toggles might not be enough to compensate for the toggles introduced by the isolation gates and nets.

Automatic Versus User-Driven Operand Isolation Insertion

Power Compiler automatically chooses which operands to isolate when you activate operand isolation. For information, see [“Specifying Operand Isolation Style and Selecting Insertion Mode” on page 8-12](#).

Automatic Versus Manual Operand Isolation Rollback

You can remove the operand isolation implementation either automatically or manually. For information, see [“Operand Isolation Rollback” on page 8-15](#).

Operand Isolation Methodology Flows

The two approaches to incorporate operand isolation into your design flow are as follows:

- [Two-Pass Approach \(Recommended\)](#)
- [One-Pass Approach](#)

Two-Pass Approach (Recommended)

The two-pass approach entails an initial compile followed by an incremental compile. This flow consists of two stages; isolation logic is inserted during the first stage, followed by timing and power analysis. Rollback can take place in the second stage.

[Figure 8-3](#) and [Figure 8-4](#) illustrate the insertion stage and the rollback stage for the two-pass flow.

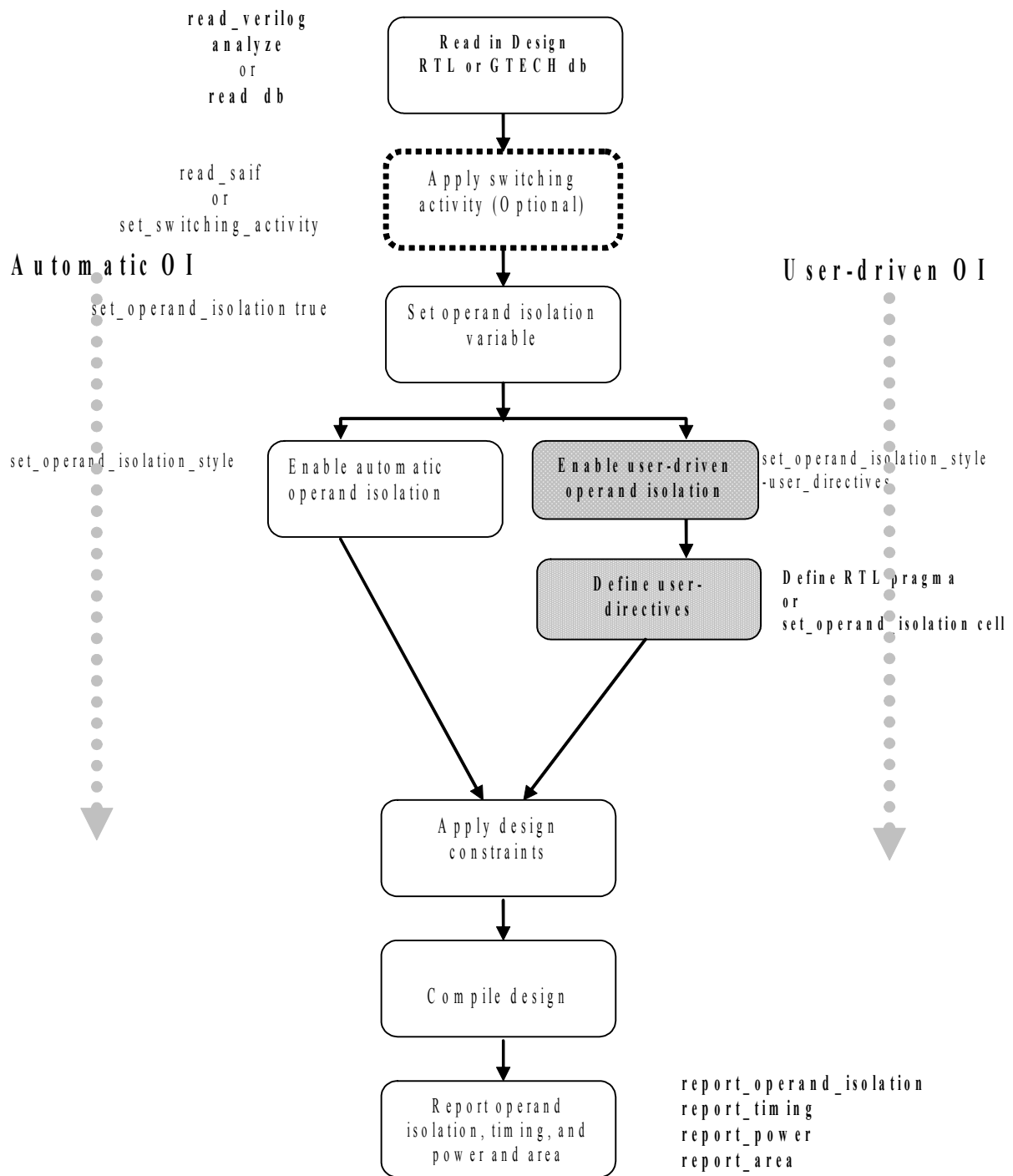
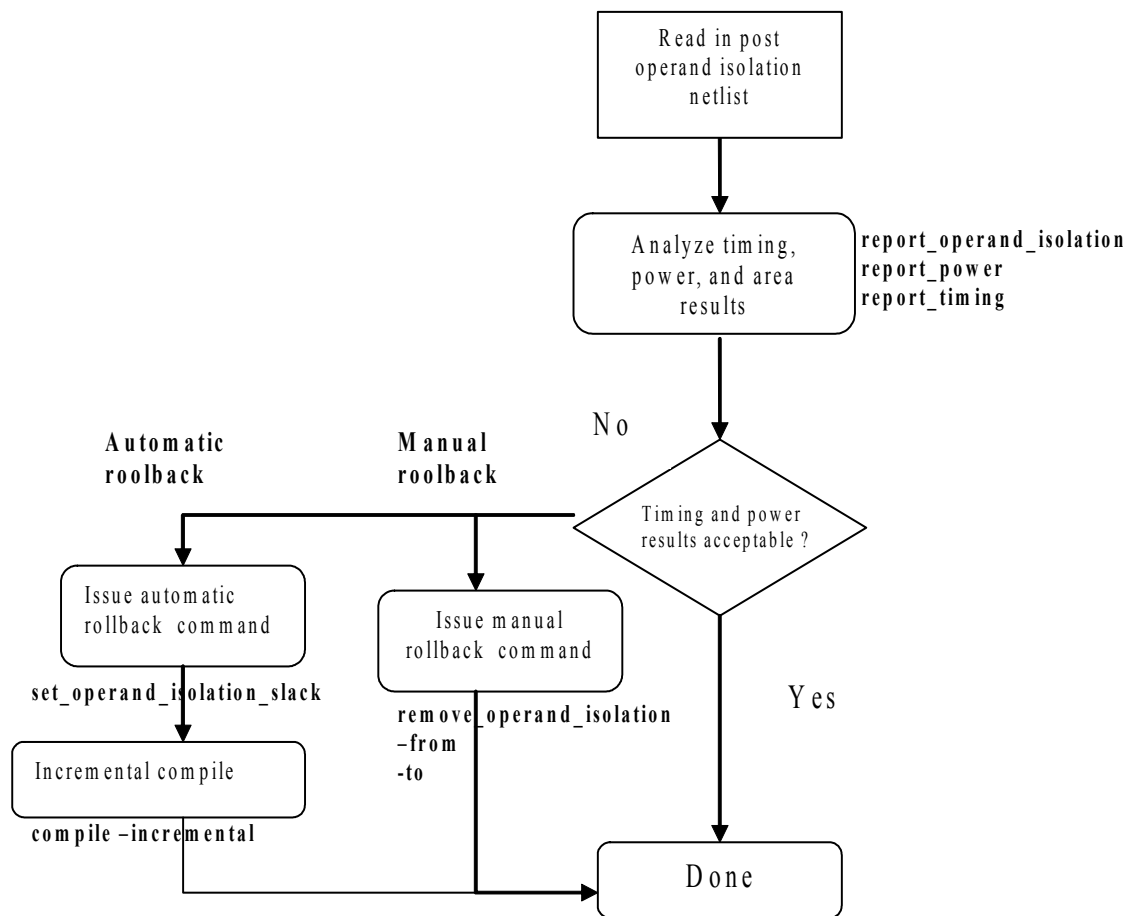
Figure 8-3 Two-Pass Approach: Operand Isolation Insertion

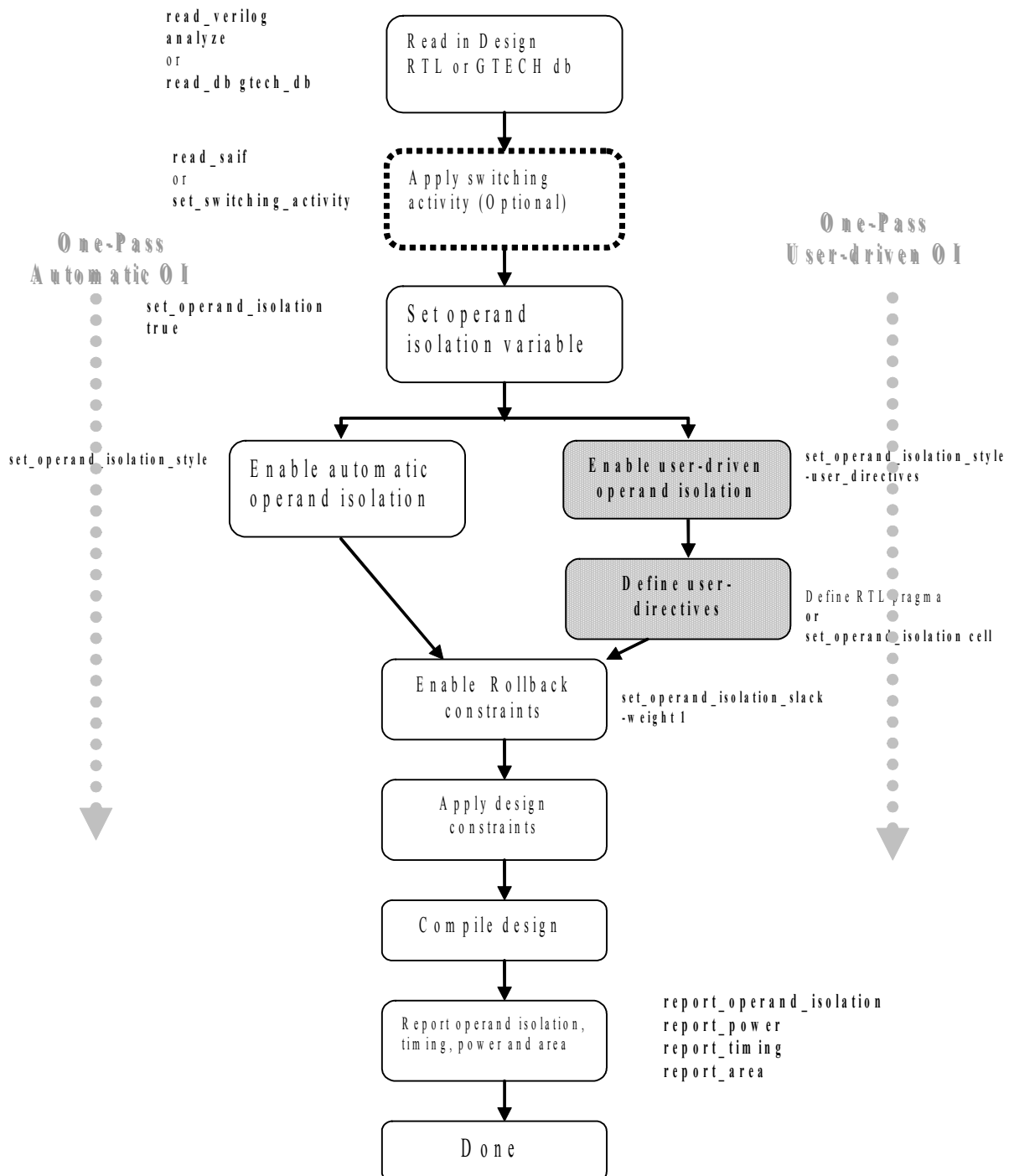
Figure 8-4 Two-Pass Approach: Analyzing Operand Isolation Results and Rollback



One-Pass Approach

The one-pass approach entails only one compile step. Operand isolation insertion is performed during the mapping stage while rollback take place during timing optimization in the same compile. [Figure 8-5](#) illustrates the one-pass flow.

Figure 8-5 One-Pass approach



Sample Scripts

The following script demonstrates a two-pass flow with automatic operand isolation:

```
# read in switching activity files
read_saif -input risc.saif -instance testrisc/proc

# issue OI constraints
set do_operand_isolation true

set_operand_isolation_style \
-logic adaptive \
-verbose

# set weight to 0 to disable rollback, default weight is 0
set_operand_isolation_slack 0.3 -weight 0

# source design constraints
source design_constr.tcl

compile_ultra

report_operand_isolation -verbose -all_objects
report_timing
report_power

set_operand_isolation_slack 0

compile_ultra -incr

report_operand_isolation -verbose -all_objects
report_timing
report_power
```

The following script demonstrates a two-pass flow with user-driven operand isolation.

```
# issue OI constraints
set do_operand_isolation true
set_operand_isolation_style \
-verbose \
-user_directives

# specify user directives
set_operand_isolation_cell [ get_cells u1/add_14 ]
set_operand_isolation_cell [ get_cells u2 ]
# set weight to 0 to disable rollback, default weight is 0
set_operand_isolation_slack 0.6 -weight 0

# source design constraints
source design_constr.tcl
```

```
compile_ultra

report_operand_isolation -verbose -all_objects
report_timing
report_power

set_operand_isolation_slack 0

compile_ultra -incr

report_operand_isolation -verbose -all_objects
report_timing
report_power
```

The following script demonstrates a one-pass flow with automatic operand isolation.

```
# issue OI constraints
set do_operand_isolation true

set do_operand_isolation true
set_operand_isolation_style \
    -logic adaptive \
    -verbose \

# set weight to a non-zero number to enable auto-rollback
set_operand_isolation_slack 0.3 -weight 1

# source design constraints
source design_constr.tcl

compile_ultra

# report and analysis
report_operand_isolation -verbose -all
report_timing
report_power
```


Commands and Variables Related to Operand Isolation

This section contains the summary of operand isolation related commands and variable. For a more detailed explanation, see [“Using Operand Isolation” on page 8-12](#).

Table 8-1 Operand Isolation Related Commands

Command Syntax	Usage	Example
set_operand_isolation_style [-logic AND OR adaptive] [-user_directives] [-verbose]	1. Specify the logic type to be used for operand Isolation. 2. Enable or disable user-driven operand isolation.	set_operand_isolation \ -logic adaptive \ -verbose
set_operand_isolation_cell <object_list> [flag]	Define the cell object for operand isolation.	set_operand_isolation_cell \ [get_cells u1/add_16] \ true
set_operand_isolation_scope <object_list > [flag]	Define the operand isolation scope	set_operand_isolation_scope \ [get_design foo] \ false
set_operand_isolation_slack [<float>] [-weight <float>]	Define the timing threshold to enable automatic operand isolation rollback	set_operand_isolation_slack \ 2.0 \ -weight 0.1
remove_operand_isolation [-from <from_list>] [-to <to_list>]	Manually remove the operand isolation logic inserted by Power Compiler	remove_operand_isolation \ -from u1/u2/EN \ -to u1/u2/out_reg[0]/D
report_operand_isolation [-instances] [-isolated_objects] [-unisolated_objects] [-all_objects] [-verbose] [-no_hier] [-nosplit] [<object_list>]	Report the status of operand isolation in the current design.	report_operand_isolation \ -all_objects \ -verbose

Table 8-2 Operand Isolation Related Variables

Variable	Usage	Example
do_operand_isolation	Setting this variable to “true” enables operand isolation. Default value is “false”	set do_operand_isolation true

Using Operand Isolation

The following topics are essential for applying operand isolation in your design flow:

- [Specifying Operand Isolation Style and Selecting Insertion Mode](#)
- [Controlling the Scope for Operand Isolation](#)
- [Defining User Directives](#)
- [Operand Isolation Rollback](#)
- [Operand Isolation Reporting](#)

Specifying Operand Isolation Style and Selecting Insertion Mode

Before starting operand isolation, specify the isolation style. The `set_operand_isolation_style` command is used to specify the logic type of the isolation gate and provide an option to enable user-driven operand isolation mode.

When you issue this command without any option, the automatic operand isolation mechanism takes place. The user-driven mode is enabled by specifying the `-user_directives` option. The syntax is

```
set_operand_isolation_style
    [-logic AND | OR | adaptive]
    [-user_directives] [-verbose]
```

For information about this command, see the man page.

Controlling the Scope for Operand Isolation

By default, Power Compiler traverses the netlist in a top-down fashion and all the designs are processed for operand isolation. However, you can utilize the `set_operand_isolation_scope` command to specify whether to include or exclude certain hierarchies for operand isolation processing. The syntax is

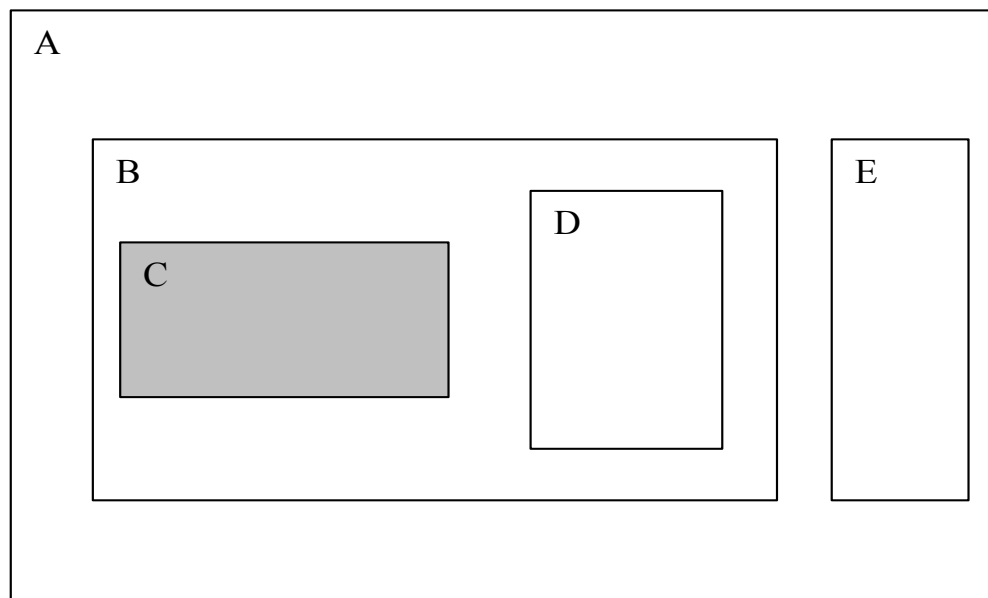
```
set_operand_isolation_scope object_list
[ true | false ]
```

The concept of the `set_operand_isolation_scope` command is similar to the `set_dont_touch` or `set_boundary_optimization` command. This command sets an attribute on the `object_list` to control whether or not to enable (the option is set to `true`) or disable (the option is set to `false`) the processing of `object_list` for operand isolation.

If a design or a hierarchical cell does not have the operand isolation scope attribute specified, the behavior is inherited from that of the parent instance. By default, the attribute of the top level is true, therefore, all the sub-designs are also true, which means operand isolation processing is enabled for the entire design.

The following example illustrates how to change the default scope behavior. In [Figure 8-6](#), the hierarchy relationships are as follows: A is the top level design. Therefore, A is the parent instance of hierarchical cell B and E, and B is the parent instance of hierarchical cell C and D.

Figure 8-6 `set_operand_isolation_scope` Example



In run script, if you issue

- Case 1:

```
set_operand_isolation_scope [ get_designs A ] false
```

None of the designs are processed for operand isolation.

- Case 2:

```
set_operand_isolation_scope [get_designs A] false
set_operand_isolation_scope [get_cells C] true
```

Only hierarchical cell C (shown in the shaded area) is processed for operand isolation.

- Case 3:

```
set_operand_isolation_scope [get_cells B] false
set_operand_isolation_scope [get_cells C] true
```

Hierarchical cells C and E are processed for operand isolation.

Defining User Directives

The `set_operand_isolation_cell` command specifies the operand isolation user directive onto the objects in the design. It sets attributes on the candidates to be included or excluded for operand isolation. The syntax is

```
set_operand_isolation_cell object_list [true|false]
```

where the object list contains arithmetic operators or hierarchical combinational cells.

- When the option is set to `true` and the user-driven operand isolation is performed, Power Compiler inserts isolation gates for the objects in *object_list* that meet the ODC conditions.
- When the option is set to `false`, Power Compiler excludes all objects in *object_list* from operand isolation. Both automatic and user-driven operand isolation mode honor this directive.

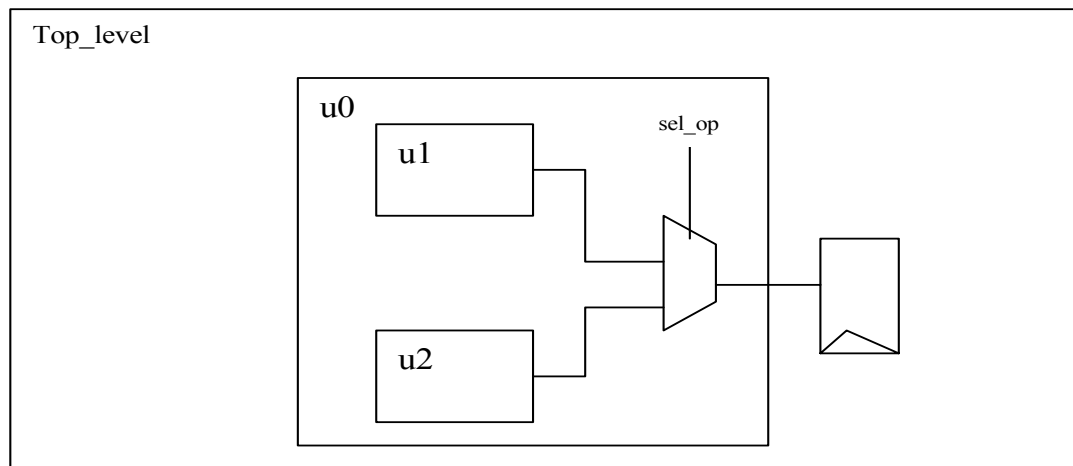
The example in [Figure 8-7](#) illustrates a scenario when user-driven operand isolation mode is applied. The design includes the hierarchical cells: `u0`, `u0/u1` and `u0/u2`.

In the run script, if you issue

```
set_operand_isolation_style -user_directives
set_operand_isolation_cell [ get_cell u0 ] false
set_operand_isolation_cell [ get_cell u0/u1 ] false
set_operand_isolation_cell [ get_cell u0/u2 ] true
```

The hierarchical cells `u0/u2` are isolated and `u0/u1` is not isolated.

Figure 8-7 The `set_operand_isolation` Command Example



For more information, see [“Examples of Using the `set\_operand\_isolation\_scope` and `set\_operand\_isolation\_cell` Commands” on page 8-25.](#)

Operand Isolation Rollback

There are two ways to perform rollback operand isolation implementation: automatically and manually.

Automatic Rollback Mechanism

To perform automatic rollback operand isolation, specify the following command

```
set_operand_isolation_slack slack_number -weight num
```

The default slack number and weight are both 0. For weight, you can specify a number between 0 and 1.

This command sets the targeted timing threshold for the worst negative slack to control the automatic rollback operation.

In a one-pass operand isolation flow with automatic rollback, the insertion of operand isolation logic occurs in the initial mapping phase. During timing optimization phase, Power Compiler evaluates the threshold and the weight value specified by the `set_operand_isolation_slack` command and then compares it to the worse negative slack to determine whether or not to rollback.

However, since the worse negative slack at this point does not represent the final worse negative slack once all the timing optimizations are complete, better results can be obtained by relaxing these constraints. Otherwise, Power Compiler would be too pessimistic and remove non-violating operand isolation logic.

The `-weight` option relaxes the operand isolation slack constraint. When the weight is set to 0 (the default) it disables automatic rollback during the initial compile step. For the one-pass flow, the weight number should be set to a number between 0 and 1. If the weight is set to 1, Power compiler honors the exact slack number for the threshold. Setting the number between 0 and 1 relaxes the slack constraint and gives Power Compiler more opportunities to preserve the isolation logic. This allows Power Compiler to control the trade-off between timing and power optimization.

At the end of the compile, if the timing result is not acceptable, invoke either the manual or automatic rollback mechanism with an additional incremental compile to remove the isolation logic on the timing critical path. This is considered a two-pass approach.

In a two-pass operand isolation flow with automatic rollback, set the weight to 0 for the initial compile. For the subsequent incremental compile, you can adjust the slack number. Otherwise, Power Compiler utilizes the slack number from the previously issued `set_operand_isolation_slack` command to perform rollback operation. Note that Power Compiler does not relax the slack constraint during incremental compile (i.e., the `-weight` option is ignored).

Manual Rollback Mechanism

Manual rollback is achieved with the `remove_operand_isolation` command. The syntax is

```
remove_operand_isolation [ -from < starting_point> ]  
                        [ -to <end_point>]
```

The manual rollback mechanism is available in a two-pass operand isolation flow. By issuing this command, Power Compiler only removes the isolation logic of the timing paths specified between the start and endpoints while preserving the rest of the operand isolation logic followed by an incremental compile. Note that you must specify at least one of the `[-from]` or `[-to]` options. These options remove the isolation logic regardless of the slack value specified by `set_operand_isolation_slack` command.

Sample Scripts for Operand Isolation Rollback

Two-pass operand isolation flow with automatic rollback:

```
set do_operand_isolation true  
set_operand_isolation_style \  
    -logic adaptive \  
    -verbose \  
    -weight 0
```

```
# Apply design timing constraints here

# Disable automatic rollback here
set_operand_isolation_slack 0.7 -weight 0

compile_ultra

report_power
report_timing
report_operand_isolation -verbose -all

# this removes operand isolation logic on the timing critical paths if
# the WNS at this point is worse than 0.7

compile_ultra -incr

report_timing
report_power
```

Two-pass operand isolation flow with manual rollback:

```
set do_operand_isolation true

set do_operand_isolation true
set_operand_isolation_style \
    -logic adaptive \
    -verbose \

# Apply design constraints here

# Does not automatic remove OI here
set_operand_isolation_slack 0.4 -weight 0

compile_ultra

report_operand_isolation -verbose -all

report_timing

# Apply manual OI removal constraint here
remove_operand_isolation -from EN -to z_reg[0]/D

report_timing
report_power
report_operand_isolation -all -verbose
```

One-pass operand isolation flow with automatic rollback:

```
set do_operand_isolation true
set do_operand_isolation true
set_operand_isolation_style \
    -logic adaptive \
```

```

-verbose \

# Apply design timing constraints here

# Enable automatic rollback here
set_operand_isolation_slack 0.7 -weight 0.8

compile_ultra

report_power
report_timing
report_operand_isolation -verbose -all

```

Operand Isolation Reporting

The `report_operand_isolation` command is used for the final operand isolation report. The command syntax is:

```

report_operand_isolation
    [-instances]
    [-isolated_objects]
    [-unisolated_objects]
    [-all_objects]
    [-verbose]
    [-no_hier]
    [-nosplit]
    [object_list]

```

[*object_list*] is the collection of isolated object list. For option information, see the man page.

By default, without specifying any option, Power Compiler only prints out the operand isolation summary table.

An example of an operand isolation verbose report:

```

-----
Isolated Objects Report
-----
Parent Instance: <top_level>
Isolated Object: add_14
  Object Type: operator
    Style: adaptive
    Method: user
  Control Signal: n36
  Gate Count : 16
Original Data Net      Isolated Pin      Isolation
Gate      Type
-----
c[0]      add_14/B[0]      C34
AND

```


c[1]	add_14/B[1]	C33	
AND			
c[2]	add_14/B[2]	C32	
AND			
c[3]	add_14/B[3]	C31	
AND			
c[4]	add_14/B[4]	C30	
AND			
c[5]	add_14/B[5]	C29	
AND			
c[6]	add_14/B[6]	C28	
AND			
c[7]	add_14/B[7]	C27	
AND			
b[1]	add_14/A[1]	C25	
AND			
b[2]	add_14/A[2]	C24	
AND			
b[3]	add_14/A[3]	C23	
AND			
b[4]	add_14/A[4]	C22	
AND			
b[5]	add_14/A[5]	C21	
AND			
b[6]	add_14/A[6]	C20	
AND			
b[7]	add_14/A[7]	C19	
AND			
b[0]	add_14/A[0]	C26	OR

Note that the `-verbose` option needs to be issued along with `-all_objects` or `-isolated_objects`. Otherwise, you must provide the isolated *object_list*.

You can see more examples in [“Operand Isolation Summary Report” on page 8-27](#).

Interoperability

The following sections show how operand isolation works with Power Compiler and other Synopsys tools.

Operand Isolation and Clock Gating

Operand isolation works with clock-gated designs. Power Compiler is able to automatically extract the control signal used for operand isolation.

Specify the following in the run script to enable clock gating and operand isolation:

```

set_clock_gating_style
insert_clock_gating
set_do_operand_isolation true
set_operand_isolation_style -logic adaptive -verbose
compile_ultra

```

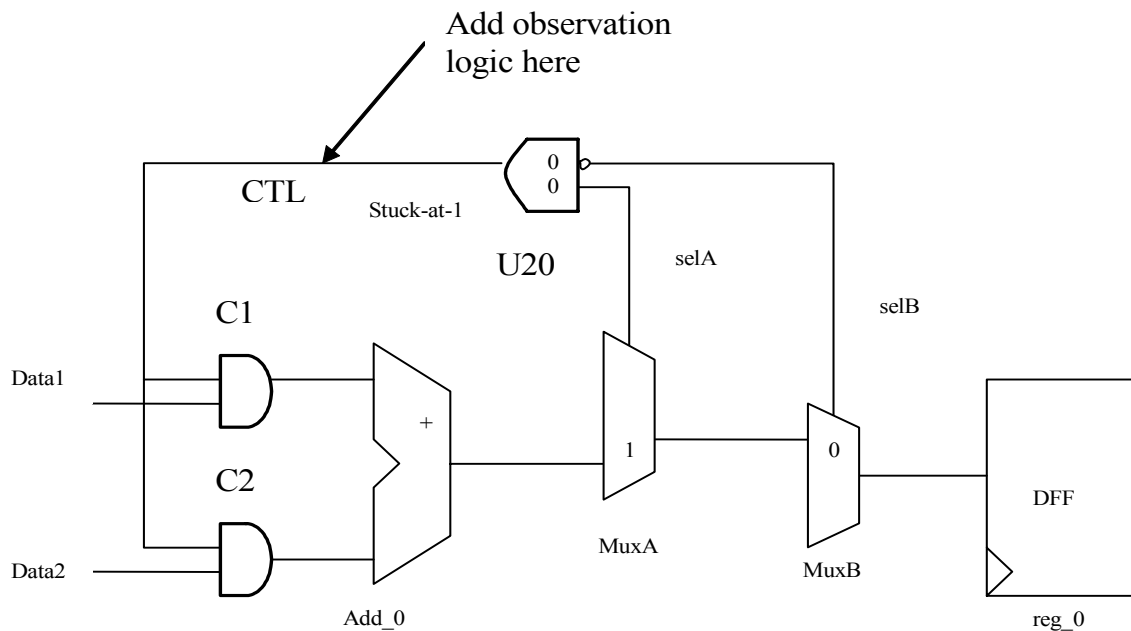
Operand Isolation and Testability

The testability issue is related to the isolation gate type implemented in the design. If the isolation gate type is AND, the “stuck-at-1” fault of the control signal cannot be observed. As shown in [Figure 8-8](#), to detect the “stuck-at-1” fault at the control signal CTL, both inputs of the AND gate U20 need to be 0, therefore, selB needs to be 1, and selA needs to be 0. However, in this configuration, the output of the operator is not observable by the register. In other words, we cannot observe “stuck-at-1” fault at CTL when isolation logic is AND style.

Similarly, if the isolation gate type is OR, the “stuck-at-0” fault of the control signal is not observable.

To solve this problem, label the control signal as an explicit observation point for DFT Compiler to generate the observation circuits.

Figure 8-8 Adding Observation Point to Activation Signal in AND Style Isolation Logic



Debugging Tips

This section describes the possible reasons why operand isolation is not implemented in either automatic or user-driven mode.

- The operator does not have an observable don't care (ODC) condition. The outputs of the operator (or combinational hierarchical cell) are always used.
- In automatic operand isolation mode, if there is no significant power reduction by inserting operand isolation for the potential objects, no isolation is implemented. For example, the size of the operator is too small or not complex enough or the output of the operator is observed most of the time.
- In RTL netlist, if an input data net is tied to a constant value or there is zero switching activity on the input data net, no isolation is implemented on that net.
- There are don't touch attributes set on the parent instance of the operator.
- The object specified by the `set_operand_isolation_cell` command contains sequential elements.
- There is a `set_case_analysis` constraint onto the control signal. Depending on the case constraint, operand isolation might not be implemented.

If you specify the following constraint in the run script

```
set_case_analysis 0 [get_ports EN]
```

This command specifies the constant value 0 to port EN. In the Verilog netlist, if the output of the operand is always observed under this configuration, no operand isolation opportunity is available.

- Resource sharing occurs between the operators. In user-driven mode, some of user directive operators (specified by `set_operand_isolation_cell`) don't get isolated due to the fact that the sharing resource of these operators might not have ODC condition available.

In this Verilog netlist, there are three adders inferred, their ODC conditions are

Operator	ODC
Inst2b/add_6	!en
Inst2c/add_6	en
Inst2d/add_6	clear + en

If we flatten the design and specify `set_operand_isolation_cell` onto these adders during compile, and if there is a resource sharing for the three adders, the conjunction of the individual operator's ODC condition is an empty set. Therefore, no operand isolation is implemented.

Examples

The following sections contain examples of operand isolation usage.

Verilog RTL With Observable Don't Care Conditions

All the following Verilog RTL netlists infer observable don't care control constructs at the fanout of the arithmetic operators:

The following example shows observable don't care conditions at the adder output. The observable don't care sets are derived from inputs selA, selB, and clear.

```
module test(a,b,c,clk,clear,selA,selB,out);
    input [7:0] a,b,c;
    input clk;
    input clear,selA,selB;
    output [7:0] out;

    reg [7:0] out;

    wire [7:0] comb, reg_in;

    assign comb = a +b ;
    assign reg_in = selB ? c : selA ? comb : a ;

    always @(posedge clk)
        begin
            if (~clear)
                out <= 8'b0;
            else
                out <= reg_in;
        end

endmodule
```

The following example shows observable don't care conditions at the combinational module output. observable don't care sets are derived from inputs sel and EN.

```
module test(a,b,c,clk,EN,sel,out);
    input [7:0] a,b,c;
    input clk;
    input EN,sel;
    output [7:0] out;
    wire [7:0] comb_wire;

    assign comb_wire = sel ? a : (b+c);
    assign out = EN ? 8'b0 : comb_wire;
```

```
endmodule
```

The following example shows observable don't care conditions at the adder output inferred by enable type of flop. observable don't care sets are derived from inputs sel and EN.

```
module test(a,b,c,clk,EN,out);
    input [7:0] a,b,c;
    input clk;
    input EN;
    output [7:0] out;
    wire [7:0] comb_wire;
    reg [7:0] out;

    assign comb_wire = b+c ;

    always @(posedge clk)
        begin
            if (EN)
                out<= comb_wire;
            end
        endmodule
```

Report Operand Isolation Progress

The operand isolation progress is displayed when the `-verbose` option is specified with the `set_operand_isolation_style` command.

```
Beginning Pass 1 Mapping
```

```
-----
Processing 'adder'
Processing 'comb'
Processing 'seq'
Processing 'top'
```

```
Updating timing information
```

```
Information: The target library(s) contains cell(s), other
than black boxes,
that are not characterized for internal power. (PWR-24)
```

```
Beginning Implementation Selection
```

```
-----
Processing 'adder_0_DW01_add_9_0'
Processing 'comb_0_DW01_add_8_0'
Processing 'adder_1_DW01_add_9_0'
Processing 'comb_1_DW01_add_8_0'
Processing 'adder_2_DW01_add_9_0'
```

```
Information: Performing operand isolation on design 'top'
Information: Propagating switching activity (low effort zero
delay simulation).
```

(PWR-6)

Warning: Design has nonannotated primary inputs. (PWR-414)

Warning: Design has nonannotated sequential cell outputs.

(PWR-415)

ISOL.	ISOLATED	UNISOLATED	OI	PARENT INSTANCE	
GATES	OPER.	HIER.	OPER.	HIER.	APP
0	0	0	0	1	N <top level>
0	0	0	0	0	- seq_inst
0	0	0	1	0	N top_inst_adder
16	1	0	0	0	Y u2
16	1	0	0	0	Y u1
32	2	0	1	1	Y

The following example shows the progress of an user-driven operand isolation flow. Note that the PWR-519 warnings occur when Power Compiler could not find the operand isolation opportunity for the objects specified by the `set_operand_isolation_cell` command or RTL pragma.

ISOL.	ISOLATED	UNISOLATED	OI	PARENT INSTANCE	
GATES	OPER.	HIER.	OPER.	HIER.	APP
0	0	0	0	1	N <top level>
Warning: No operand isolation applied to cell 'inst2c/add_6' because no opportunity for isolation was found. (PWR-519)					
0	0	0	1	0	N inst2c
Warning: No operand isolation applied to cell 'inst2b/add_6' because no opportunity for isolation was found. (PWR-519)					
0	0	0	1	0	N inst2b
32	0	1	0	1	Y inst2a
0	0	0	0	0	- inst2a/inst3a
Warning: No operand isolation applied to cell 'inst2a/inst3a/inst4a/add_6' because no opportunity for isolation was found. (PWR-519)					
0	0	0	1	0	N inst2a/inst3a/inst4a
Warning: No operand isolation applied to cell 'inst2a/inst3a/inst4b/add_6' because no opportunity for isolation was found. (PWR-519)					
0	0	0	1	0	N inst2a/inst3a/inst4b
32	0	1	4	2	Y

Examples of Using the `set_operand_isolation_scope` and `set_operand_isolation_cell` Commands

The following example illustrates the usage of the `set_operand_isolation_cell` and `set_operand_isolation_scope` commands.

The Verilog netlist:

```
module test(a,b,c,clk,clear,sel_op,EN_1, EN_2,z);
    input [7:0] a,b,c;
    input clk;
    input sel_op,EN_1,EN_2;
    input clear;

    output [7:0] z;
    reg [7:0] z;

    wire [7:0] d1,d2;

    comb u1(a,b,EN_1,d1);
    comb u2(c,b,EN_2,d2);

    always @(posedge clk or negedge clear)
        begin
            if (~clear)
                z <= 8'b0;
            else
                z <= sel_op ? d1 : d2;
        end
endmodule

module comb(a,b,EN,z);
    input [7:0] a,b;
    input EN;
    output [7:0] z;

    assign z = EN ? ( a + b ) : b ;
endmodule
```

The user-driven mode is activated in this example. First, read the netlist in `dc_shell` and perform the following:

```
dc_shell> current_design comb
Current design is 'comb'.
{comb}

dc_shell> get_cells *
{add_34, C16, B_0, B_1, I_0, B_2}
```

```
dc_shell> get_cell -hier *add*
{u1/add_34 u2/add_34}
```

If you want to isolate only `u1/add_34`, issue the following commands in the run script:

```
set_operand_isolation_scope [ get_cell u2 ] false
set_operand_isolation_cell [ get_cell u1/add_34 ] true
```

In this case, Power Compiler does not process the hierarchical cell `u2` and anything underneath `u2` for operand isolation. Meanwhile, it inserts isolation only onto `u1/add_34`.

During the compile, `u2` is excluded from the operand isolation process and it does not show up in operand isolation status:

ISOL.	ISOLATED	UNISOLATED	OI						
GATES	OPER.	HIER.	OPER.	HIER.	APP			PARENT	INSTANCE
	0	0	0	0	0	-	<top level>		
	16	1	0	0	0	Y	u1		
	16	1	0	0	0	Y			

You can verify from the final report that only `u1/add_34` were isolated.

```
Isolated Objects Report
-----
Parent Instance: u1
Isolated Object: add_34
  Object Type: operator
    Style: adaptive
    Method: user
  Control Signal: EN
    Gate Count : 16
Original Data Net      Isolated Pin      Isolation
Gate      Type
-----
a[0]      add_34/A[0]      C8
AND
b[0]      add_34/B[0]      C16
AND
b[3]      add_34/B[3]      C13
AND
b[2]      add_34/B[2]      C14
AND
a[2]      add_34/A[2]      C6
AND
a[3]      add_34/A[3]      C5
AND
b[5]      add_34/B[5]      C11
AND
```


a[5]	add_34/A[5]	C3
AND		
b[1]	add_34/B[1]	C15
AND		
a[1]	add_34/A[1]	C7
AND		
b[4]	add_34/B[4]	C12
AND		
a[4]	add_34/A[4]	C4
AND		
b[6]	add_34/B[6]	C10
AND		
a[6]	add_34/A[6]	C2
AND		
b[7]	add_34/B[7]	C9
AND		
a[7]	add_34/A[7]	C1
AND		

----- Unisolated Objects Report

Unisolated Object	Object Type
u2	hierarchical
u1	hierarchical
u2/add_34	operator

Operand Isolation Summary Report

report_operand_isolation

```
*****
Report : isolation
Design : top
Version: W-2004.12
Date   : Tue Oct 26 16:00:38 2004
*****
```

Library(s) Used:

slow (File: slow.db)

Operand Isolation Summary

Isolation Style	adaptive
Isolation Method	automatic
Number of Isolation gates	32

Number of Isolated objects	2 (20.00%)
operators	2 (20.00%)
hierarchical cells	0 (0.00%)
Number of Unisolated objects	8 (80.00%)
operators	3 (30.00%)
hierarchical cells	5 (50.00%)

9

Gate-Level Power Optimization

Power Compiler optimizes your designs for power. During an optimization session, Power Compiler performs additional steps to optimize your design for dynamic and leakage power.

This chapter contains the following sections:

- [Overview](#)
- [General Gate-Level Power Optimization](#)
- [Leakage Power Optimization](#)
- [Dynamic Power Optimization](#)

Power Compiler always works within the Design Compiler shell and is transparent to Design Compiler users. This feature enables seamless integration of power optimization into your synthesis environment. Working within the Design Compiler shell, Power Compiler can optimize for power while monitoring time and area cost functions.

Before reading this chapter, familiarize yourself with the basic concepts of synthesis and optimization as found in the Design Compiler documentation. Power Compiler optimizes your design for power if you have set power constraint on your design.

Technology libraries characterized for power are required for power optimization. Using fully characterized libraries that contain a variety of cells with different drive strength characteristics, you can realize average dynamic power and leakage power reductions with multivoltage threshold libraries compared to designs optimized for timing and area only.

Overview

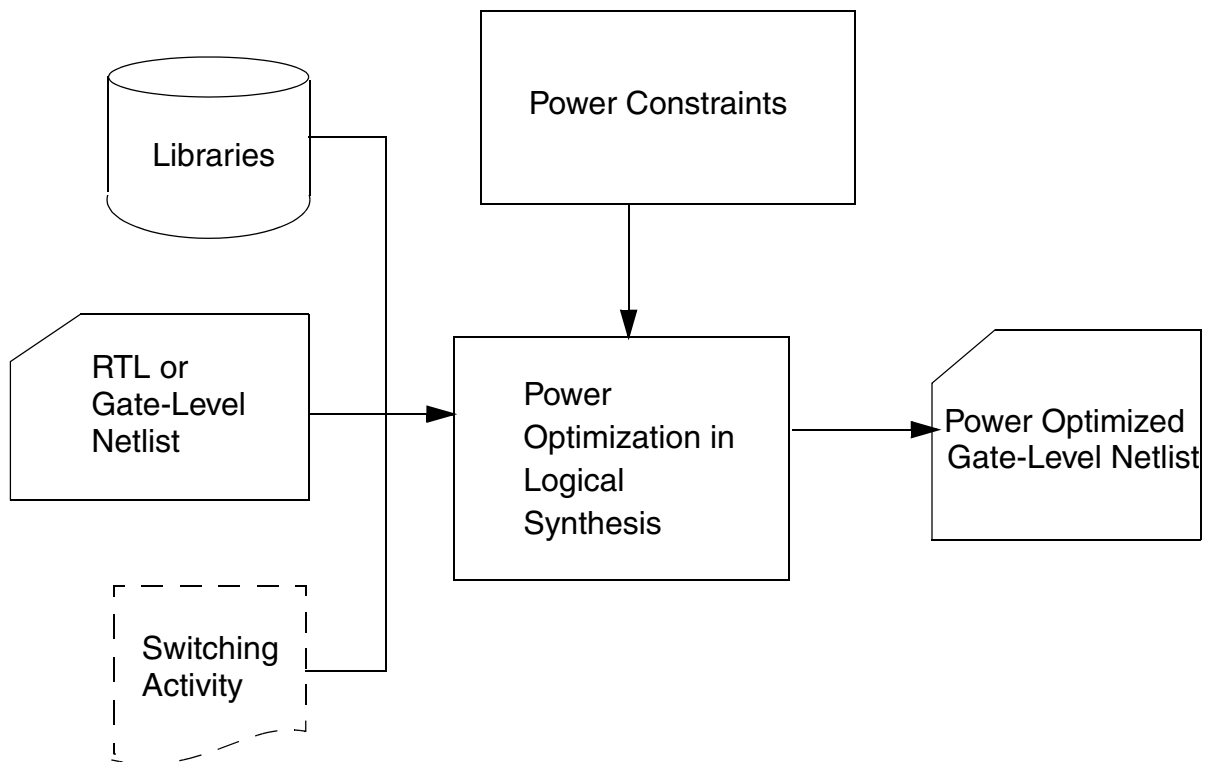
The speed of the transistor continues to improve. The most common technique used to achieve the high performance is to reduce the geometry of the transistor as well as the voltage to operate it. To maintain the speed and noise margin of the smaller transistor, the threshold voltage needs to be lowered too. Since the threshold voltage has exponential impact on the transistor leakage power, low threshold voltage transistors have high leakage power. Minimizing the leakage power is one of the major challenges to be resolved, especially in lower geometries.

In any design, there are critical and non-critical timing paths. Using a lower speed cell on non-critical path does not affect the performance of a design. A slower cell allows higher threshold voltage, which reduces leakage power dramatically. Optimizing the high speed and low speed cells on different timing paths leads to a balanced design with high performance and low leakage power.

Input and Output of Power Optimization

Figure 9-1 illustrates the flow for gate-level power optimization.

Figure 9-1 I/O Flow for Power Optimization



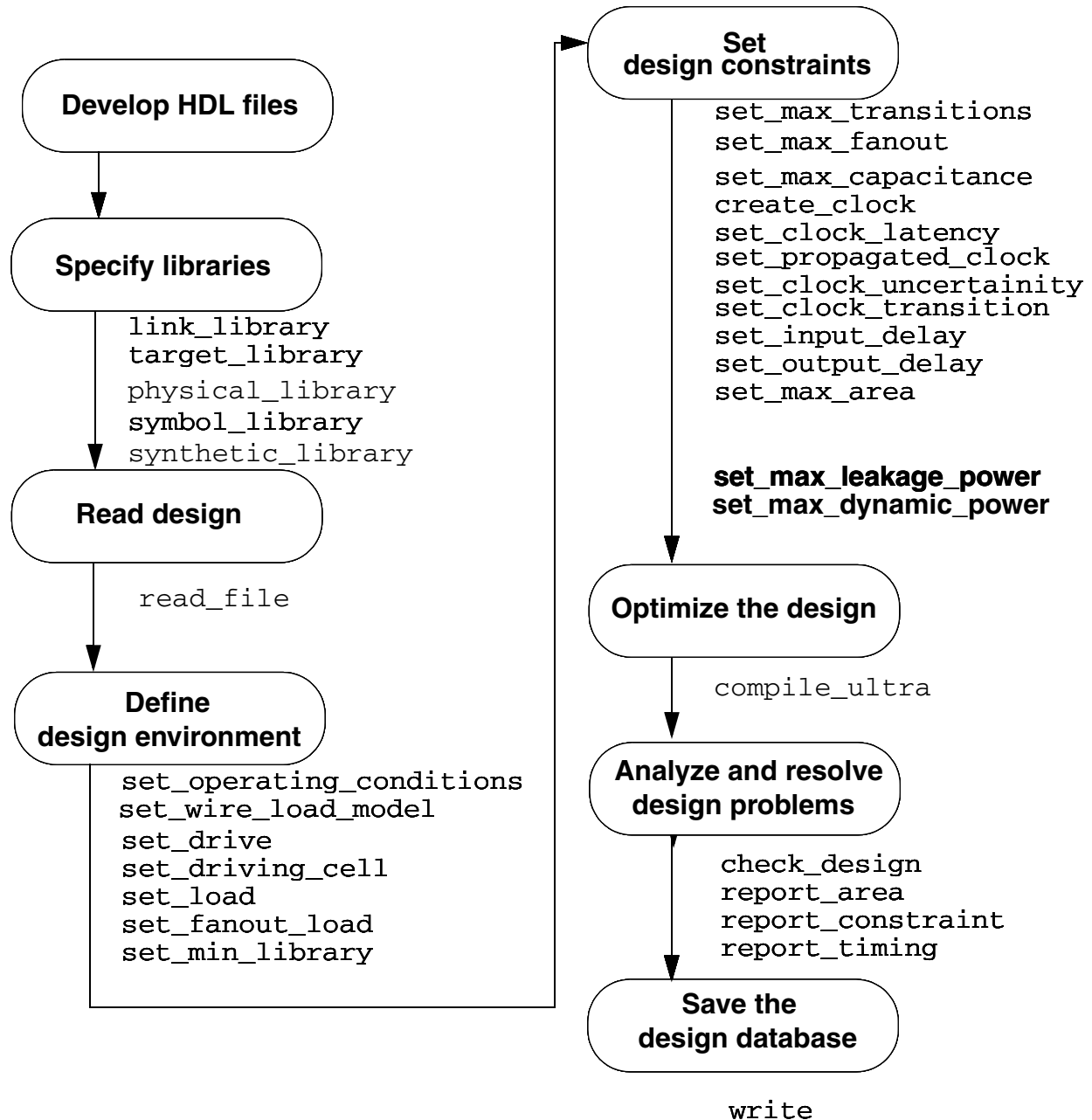
The inputs for gate-level power optimization are:

- RTL or gate-level netlist and floor plan (optional)
This netlist is not power optimized.
- Power constraints
Power constraints set the target power value for optimization. Optimization options specify different algorithms and conditions.
- Libraries
Power Compiler selects different library cells to rebuild the netlist with the optimized power. Multivoltage threshold libraries are highly recommended for leakage optimization.
- Switching activity
This is required for dynamic and total power optimization, and is used for high accuracy in leakage optimization.

The output of gate-level power optimization is a new gate-level netlist that has optimized power. The optimization is implemented with the `compile` or `compile_ultra` commands.

Power Optimization in Synthesis Flow

Figure 9-2 Flow for Synthesis Power Optimization



General Gate-Level Power Optimization

The following sections describe how to perform power optimization with Power Compiler and how to use constraints.

Power Optimization Commands

Power optimization is performed together with other optimizations. Commands that start optimizations include:

- `compile` and `compile_ultra` in Design Compiler
- `compile_ultra` in Design Compiler topographical
- `compile_ultra -incremental` in Design Compiler

When power constraints are specified, `compile` optimizes power together with timing and area.

Power Constraints

Power constraints are set by the following commands:

- `set_max_leakage_power`
- `set_max_dynamic_power`

The set power constraint commands set attributes on the current design with the targeted power value and unit, as follows:

- `set_max_leakage_power` adds the `max_leakage_power` attribute to the current design
- `set_max_dynamic_power` adds the `max_dynamic_power` attribute to the current design

The optimization commands check the above attributes on the current design to decide if power optimization is performed. As long as the attributes exist, power optimization is performed. To stop further power optimization, these attributes need to be removed.

You can view the attribute using the `get_attribute` command:

```
get_attribute max_leakage_power [current_design]
```

The attribute can be removed by the `remove_attribute` command:

```
remove_attribute [current_design] max_leakage_power
```

Multiple power constraints can be set for the same optimization. In this case, the optimization follows the cost priority rule. For more information, see [“Cost Priority” on page 9-6](#).

If multiple `set_max` commands are specified, Power Compiler optimizes only the constraint with the highest priority. For example, if you specify both the `set_max_dynamic_power` and `set_max_leakage_power` commands, Power Compiler optimizes dynamic power and uses the leakage power constraint to ensure that the cost of optimizing leakage power remains low if you’ve set optimization effort to high.

Scope of Power Constraints

Power optimization is performed only when the power constraint has been set to the current design. Power constraint triggers the power optimization of the current design and all sub-designs.

If the power constraint is not on the current design, for example, if it is on a sub-design, no power optimization is performed on the current design and all its sub-designs. The constraint is ignored.

Design Rule Constraints and Optimization Constraints

Design Compiler calculates two separate cost functions: one for design rule constraints and one for optimization constraints. Design Compiler uses the cost functions to weigh the relative benefit of potential optimization changes to your design.

Design rule constraints and optimization constraints set the limits within which the cost functions assess potential optimization changes.

The Design Compiler documentation contains more information about design rule constraints and optimization constraints.

Cost Priority

During the first phase of mapping, Design Compiler works to reduce the optimization cost function and the design rule cost function. Each function is evaluated during compile to determine whether a change to the design would improve the total cost. Although design rule constraints are ultimately more important than delay constraints, Design Compiler and Power Compiler consider delay constraints most important during the first phase of mapping.

The full optimization cost function takes into account the following factors, listed in order of priority. Some of these components might not be active on a design:

- Design rule cost
- Maximum delay cost
- Minimum delay cost
- Maximum dynamic power cost
- Maximum static power cost
- Minimum porosity cost
- Maximum area cost

Cost function components are evaluated independently, in order of priority. An optimization move is accepted if it decreases the cost of one component without increasing the cost of a higher priority component.

For example,

- An optimization move that improves maximum delay cost is always accepted.
- An optimization move that improves leakage power is accepted only if maximum delay, design rule, minimum delay, maximum total power and maximum dynamic power costs do not increase.
- An optimization move that improves area is accepted only if no other costs increase.

Optimization stops when all costs are zero or no further improvements are made to the total cost function. After the initial phase of mapping, Design Compiler increases the priority of design rule costs.

Positive Timing Slack

Slack is the margin by which maximum or minimum path delay requirements are met. Positive slack indicates that the requirement is met; negative slack indicates that the requirement is violated. Power Compiler can use positive timing slack to decrease the power of your design. The more positive slack that exists, the more delay Power Compiler can accept in making choices for low-power cells.

Designs with excessively restrictive timing constraints have little or no positive slack to trade for power reductions. If Design Compiler produces a design with large positive slack, Power Compiler can often achieve a significant power reduction.

Unmet Constraints

An incremental compile uses the existing gate structure as a starting point for continued optimization. Usually, this ensures improvement of the circuit for timing, power, and area (or for other active constraints you define).

You might, however, see situations in which your power or area cost increases, even during an incremental compile. If you have a previously unmet design goal (violated constraint), a subsequent optimization session attempts to meet the violated constraint by sacrificing lower-priority design goals.

For example, you can have a violated timing constraint from a previous optimization session. In your next optimization session, moves are accepted that increase power or area if the moves improve the violated timing constraint. Power Compiler never violates a timing constraint in order to optimize for power. For additional information, see [“Incremental Optimization” on page 9-8](#).

Design Rule Fixing

Design Compiler tries to meet optimization constraints and design rule constraints but gives priority to design rule constraints because they are required for designs to function correctly. During the first phase of mapping, Design Compiler works to reduce the optimization cost function. During this phase, Design Compiler allows some optimization moves that create design rule violations.

After optimization, Design Compiler makes an additional pass to correct design rule violations. This pass is called design rule fixing.

During design rule fixing, Design Compiler can sacrifice optimization results to fix design rule violations. In the design rule fixing phase, you might see delay, power, and area (or other optimization results) increase in your design.

Incremental Optimization

It is recommended to run power optimization using incremental mode.

Incremental optimization uses the existing placement or netlist (in Design Compiler) as the start point for a new run to achieve other design goal. Incremental power optimization minimally disturbs the timing that has already been optimized. It usually produces better overall QoR. Its runtime is shorter.

Synthesizable Logic

Many designs have at least some elements that synthesis cannot affect. Examples of these elements are black box cells, such as RAM and ROM, and customized subdesigns on which you set the `dont_touch` attribute. Designs experience greater benefit from power optimization when greater amounts of logic are accessible to Power Compiler.

Leakage Power Optimization

Leakage power optimization is an additional step to timing optimization. During leakage power optimization the tool tries to reduce the leakage power of your design, without affecting the performance. To reduce the overall leakage power of the design, leakage power optimization is performed on paths that are not timing-critical. When the target libraries are characterized for leakage power and contain cells characterized for multiple threshold voltages, during the leakage power optimization, Power Compiler uses the library cells with appropriate threshold voltages to reduce the leakage power of the design.

Power Compiler updates the cost of the leakage power frequently during leakage power optimization. Power Compiler uses state-dependent information to improve the leakage power optimization.

Enabling Leakage Optimization

Leakage power optimization is enabled by setting the leakage power constraint. You use either a single threshold voltage or a multithreshold voltage library for leakage power optimization. However, multithreshold voltage libraries are more effective. In topographical mode, Power Compiler also supports leakage power optimization for multimode designs.

The `set_max_leakage_power` command sets the leakage power constraint and enables leakage power optimization.

The syntax of this command is

```
set_max_leakage_power num [unit]
```

Here is an example of using the command:

```
set_max_leakage_power 0
```

Using Multithreshold Voltage Libraries

Leakage power optimization can use single threshold voltage or multithreshold voltage libraries. However, multithreshold voltage libraries can save more leakage power.

Leakage power is very sensitive to threshold voltage. The leakage power varies from 4 to 50 times for different threshold voltages. The higher the threshold voltage, the lower the leakage power. On the other hand, timing varies from 5 to 30 percent. The lower the threshold voltage, the faster the timing. For the single-voltage library, the variance of threshold voltage and timing is of a similar magnitude.

Leakage power optimization is performed on the noncritical paths. The positive slacks are used to swap low speed and low leakage power cells.

Due to the sensitivity of leakage power and the insensitivity of timing to threshold voltage, optimization with multivoltage threshold libraries can result in much better leakage power savings.

Library Threshold Voltage Attributes

To define threshold voltage groups in the technology libraries, use the `set_attribute` command and add the following attributes:

- Library-level attribute:

```
default_threshold_voltage_group : "<string>" ;
```

- Library-cell-level attribute:

```
threshold_voltage_group : "<string>" ;
```

With these attributes, the threshold voltages are differentiated by the string you specify. When your technology library has at least two threshold voltage groups or if you have defined threshold voltage groups for your library cells using the `set_attribute` command, the candidate cells are grouped by the threshold voltage.

Leakage Optimization for Designs with Easy Timing Constraints

For designs that have strict timing constraints that must be met, you optimize for leakage power only on the non timing-critical paths, using the higher threshold-voltage cells from the multithreshold voltage libraries. When your design has a relatively easy-to-meet timing constraint, you might have a large number of low threshold-voltage cells in your design, resulting in higher leakage power consumption. One way to avoid this situation without having to change your target library settings is to use the `set_max_lvth_percentage` command to specify a very low percentage value for the lower threshold-voltage cells. For optimum results you can start with an area of 1 to 5% of the design for the low

threshold-voltage cells and gradually increase the percentage until the timing constraint is met. With this technique your design meets the timing constraint with minimal leakage power consumption.

The `set_max_lvth_percentage` command controls the percentage of lower threshold-voltage cells in your design. During optimization, use the `compile_ultra` command to ensure that this constraint is honored during synthesis. To see the percentage of the lower threshold-voltage cells in the synthesized design, use the `report_power` or the `report_threshold_voltage_group` commands.

The syntax of the `set_max_lvth_percentage` command is as follows:

```
set_max_lvth_percentage [max_lvth]
                        [-lvth_groups groups]
                        [-reset]
```

The `max_lvth` value specifies the maximum percentage of the total design area that can be occupied by the cells belonging to the low threshold-voltage group. The value you can specify for `max_lvth` is between 0 and 100.

The `-lvth_groups` option can be used to specify the list of voltage threshold groups that can be considered as low threshold-voltage group.

You can reset the constraint using the `-reset` option.

Use the `report_threshold_voltage_group` command to report the percentages of different multivoltage cells in the design.

Note:

Only the `compile_ultra` command honors the constraint set by the `set_max_lvth_percentage` command. The `compile` command does not.

In the following example, the `default_threshold_voltage_group` attribute is set to `HVT` and `LVT` on two libraries, `lib1` and `lib2`, respectively. The `report_threshold_voltage_group` command reports the ratio of cells used from different threshold-voltage groups:

In the following example, the maximum percentage of low threshold-voltage cells in the design is set to 10.

```
# Read multivoltage technology libraries
# Assuming lib2 is the low vt library
# Read the design
set_attribute [find library lib1] default_threshold_voltage_group \
HVT -type string
set_attribute [find library lib2] default_threshold_voltage_group \
LVT -type string
```

```
set_max_leakage_power 0
set_max_lvth_percentage 2 -lvth_groups {LVT}
compile_ultra
report_threshold_voltage_group
```

Choosing the Leakage Power Calculation Model

To choose the model that the tool should use to calculate the leakage power of the design, use the `set_leakage_power_model` command. The syntax of this command is as follows:

```
set_leakage_power_model [-type leakage | channel_width] \
                        [-mvth_weights leakage | channel_width] \
                        [-reset]
```

The default behavior is to use the leakage power attribute specified in the library characterized for leakage power.

To use the channel-width model, your target library should have the library-level and cell-level attributes for the threshold voltage groups and also the corresponding channel-width attributes described in the section, [“Channel-width Based Leakage Power Calculation” on page 3-6](#).

Calculating Leakage Power

Power Compiler calculates the leakage power of the design, using the following two methods:

- Leakage values in the library

The libraries characterized for leakage power contain the leakage power values for each library cell. Libraries can also contain the leakage values for all cells in the library. Power Compiler computes the total leakage power of the design by summing the leakage power of the library cells of the design. For more details see [“Leakage Power Calculation” on page 3-4](#).

- Using the channel-width values of threshold voltage groups in the library

The leakage power of a transistor is directly proportional to its channel-width. To optimize for leakage power, Power Compiler chooses library cells such that the channel-widths for the specific voltage threshold group is low. For more details see [“Channel-width Based Leakage Power Calculation” on page 3-6](#).

Sample Scripts For Leakage Optimization

Note:

The `report_power` and `report_constraint` commands always use state-dependent information to calculate leakage power. Dynamic and total power optimization always use state- and path-dependent information.

Using the Default Usage Model

The following sample script uses the default usage model for multivoltage threshold leakage optimization.

```
# Specify all multivoltage threshold libraries in one place
set target_library "hvt.db nvt.db lvt.db"
set link_library "*" $target_library

# Read the design
read_verilog rtl.v
link
# Enable leakage power optimization
set_max_leakage_power 0
compile_ultra
report_power
```

Using the Channel-Width Model

When the Technology Libraries are Characterized with Channel-Width Attributes

The following sample scripts illustrate the use of channel-width based leakage power calculation when the technology libraries are characterized with the channel-width attributes, for the standard cells.

In the following example the channel-width weights specified at the library-level are used to calculate the leakage power.

```
set_max_leakage_power 0
set_leakage_power_model -type channel_width
compile_ultra
```

In the following example the channel-width weights specified for the design are used to calculate the leakage power. Specifying the channel-width weights for the design overrides the library-level channel width weights specified in the technology library.

```
set_max_leakage_power 0
set_leakage_power_model -type channel_width \
-mvth_weights "lvt = 100 nvt = 300 hvt = 1"
compile_ultra
```

When the Technology Libraries Are Not Characterized with Channel-Width Attributes

When the technology library is not characterized with channel-width attributes you must specify these attributes on all the standard cells in the library. You use the `set_attribute` command to set the channel-width attribute on the standard cells. In the following example, each standard cell in the library has one type of threshold voltage and this is specified using the `set_attribute` command.

```
set_attribute -type string [get_lib_cell L1/BHVSX10] \
vth_channel_widths "hvt = 12.4 "
set_attribute -type string [get_lib_cell L1/BNVSX10] \
vth_channel_widths "nvt = 9.0 "
set_attribute -type string [get_lib_cell L1/BLVSX10] \
vth_channel_widths "lvt = 5.3"
...
set_max_leakage_power 0
set_leakage_power_model -type channel_width \
-mvth_weights "lvt = 100 nvt = 10 hvt = 1"
compile_ultra
```

Power Critical Range

In Design Compiler topographical, during leakage power optimization, the reduction of positive timing slack should be limited. This helps minimize problems during subsequent changes such as routing, crosstalk, and so on. You set the positive timing slack limit using the `physopt_power_critical_range` variable. The following example directs Power Compiler to only optimize timing paths where the positive slack is 0.2 or more.

```
set physopt_power_critical_range 0.2
```

For more information, see the man page.

Dynamic Power Optimization

After RTL clock gating or operand isolation, gate-level dynamic power optimization further reduces the dynamic power. Dynamic power optimization is an additional step to the timing optimization. After the optimization, your design consumes less dynamic power without affecting the performance.

Dynamic power optimization is activated by setting the dynamic power constraint. Optimizing dynamic power incrementally provides better quality of results and take less runtime. Dynamic power optimization depends on the switching activity. SAIF files affect the results.

Running Dynamic Power Optimization

The `set_max_dynamic_power` command sets the dynamic power constraint and enables dynamic power optimization. A sample script for dynamic power optimization can be found in [“Sample Scripts” on page 9-15](#).

The syntax of this command is

```
set_max_dynamic_power num [unit]
```

Here is an example of using the command with the default setting:

```
set_max_dynamic_power 0
```

The default setting has a well-balanced runtime and quality of result.

Annotating Switching Activity

Dynamic power optimization depends on switching activity. Annotating a correct switching activity file helps optimize dynamic power.

The common format of switching activity file is SAIF. The annotation can be performed in this way:

```
read_saif -input <SAIF_file> -instance <path>
```

For more information, see the man page.

The `set_switching_activity` script can be used for the same purpose as well. If no switching activity has been annotated, the default toggle rate is applied to the primary inputs and outputs of black box cells. Power Compiler propagates the default toggle rate throughout the design. The propagated toggles are used for dynamic power optimization.

Sample Scripts

This sample uses the default option setting. It is recommended for most designs.

```
# setup general environment #
set target_library      "lib.db"
set link_library        "*" $target_library

read_verilog    design.v
link
compile_ultra
```

```
# dynamic power optimization constraint
set_max_dynamic_power 0
read_saif -input my.saif -instance tb/top_inst

compile_ultra -inc
report_power
```

10

Retention Register Implementation

Using a retention register is a technique for reducing leakage current in standby mode without affecting the performance of the design during normal operation.

Power savings are obtained by instantiating multithreshold-CMOS retention registers that rely on low-threshold voltage transistors for performance during normal operation, and high-threshold voltage, low-leakage transistors for saving the register state during standby mode.

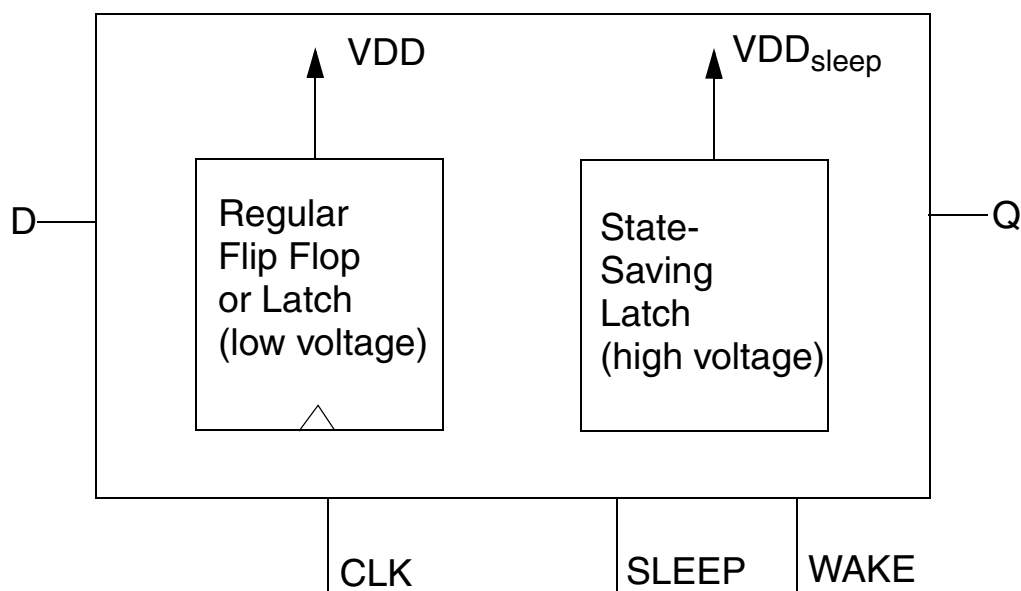
This chapter contains the following sections:

- [Multithreshold-CMOS Retention Registers](#)
- [Using Retention Registers](#)
- [Interaction With Other Synopsys Tools](#)

Multithreshold-CMOS Retention Registers

Retention cells are sequential cells that can hold their internal state when the primary power supply is shut down and that can restore the state when the power is brought up. So the retention registers are used to save leakage power in power-down applications. During normal operation, there is no loss in performance and during power-down mode, the register state is saved. These features are possible with the addition of a “balloon” latch, which holds the data from the active register. The [Figure 10-1](#) shows the basic elements of the retention register.

Figure 10-1 Retention Register Components



The retention register consists of two separate elements:

- Regular Flip-Flop or Latch

The regular flip-flop or latch consists of low-threshold voltage MOS transistors for high performance

- Retention Latch

The retention latch consists of a balloon circuit modeled with high-threshold voltage MOS transistors. It has a different power supply: VSLEEP

The behavior of these elements depends on the circuit mode. During active mode, the regular register operates at speed and the retention latch does not add to the load at the output. During sleep mode, the Q data is transferred to the retention latch, and the power

supply to the flip-flop is shut off, thus eliminating the high-leakage standby power. When the circuit is activated with the wake-up signal, the data in the retention latch is transferred to the regular register for continuous operation.

Along with the separate power supplies, additional signals such as SLEEP and WAKE are required to enable the data transfer from the regular register to the retention latch and back again, based on the mode of operation.

Based on the application, different retention register types are available to address the clocking of the data from the register to the latch and back again. Different circuit designs allow you to determine how the data transfer occurs. For example, a clock-free retention register is one type of retention register in which the state of the clock is also preserved in a retention latch. Other types of retention registers fix the clock at high or low states.

Using Retention Registers

To implement retention register in your design, follow these steps:

1. Specify the power intent for your design, using power domains.

Define the power domains, the supply network, and the primary and second power supply nets for your design. For more details see [“Power Domains” on page 12-2](#).

2. You use target libraries that contain retention registers having specific cell-level and pin-level attributes for retention operation. This information includes the power on or off behavior and register types.

3. Define your retention strategy.

The retention commands specify the strategy for inserting retention cells inside power-down domains. For more details on the retention commands see [“Defining Multivoltage Design Strategies” on page 12-17](#).

4. Map the retention registers in your design to the retention cell in the target library. Using the Power Compiler commands you can specify the constraints for the tool to map the retention register to specific retention cells in the target library. For more details, see [“Mapping the Retention Registers” on page 12-24](#).

5. You can optionally use the `check_mv_design` command to check for violations in your design. Use the `-verbose` option to get the detailed report.

6. Compile your design using the `compile_ultra` command.

During the compilation, the retention registers are mapped to the retention cells in the target library, based on the specified retention strategy.

7. Report the retention cells in your design.

The `report_retention_cell` command reports the retention cells in the design. With the `-verbose` option, this command reports the list of the retention cells, the save and restore signals and the retention strategy. For more details see the command man page.

Library Level Requirements

Retention registers consist of two elements: a regular flip-flop and a retention latch. The retention latch consists of a balloon circuit modeled with high-threshold transistors. The retention logic is generally powered up by a back-up power supply. In addition to power supplies, additional signals such as sleep and wake are required to enable the data transfer from the regular flip-flop to the retention latch and back again, based on the mode of operation. This section describes the syntax and the modeling of the retention cells in the Synopsys library file.

- **Syntax**

The Liberty library syntax for the retention cells is as follows:

```
cell(<cell_name>) {
  retention_cell : <retention_cell_style>;
  pin(<pin_name>) {
    retention_pin(<pin_class>, <disable_value>);
    ...
  }
  ...
}
```

More details of the cell-level attribute `retention_cell` and pin-level attribute `retention_pin` are described below.

- **Cell-level Attributes**

The cell-level `retention_cell` simple attribute specifies the type of the retention register. There can be multiple types of retention registers for a given register cell that provide the same functionality in normal mode but have different sleep and wake signals or clock-gating schemes. For example, if a standard D flip-flop supports two power-gating strategies, such as the following:

- type1 (data is transferred during the rising edge of the clock)
- type2 (data is transferred during the falling edge of the clock)

Then the two cell names must differ, such as `DFF_type1` and `DFF_type2`.

- **Pin-level Attributes**

The `retention_pin` complex attribute identifies the retention pins of a retention cell.

Note:

The `power_gating_pin` attribute is replaced by the `retention_pin` attribute.

However, the libraries with the older syntax are supported for backward compatibility.

The `retention_pin` attribute defines the `pin_class`. The following values are valid for `pin_class`:

- `restore` - Restores the state of the cell
- `save` - Saves the state of the cell.
- `save_restore` - Saves and restores the state of the cell.

The `disable_value` defines the integer value of the retention pin when the cell works in the normal (power up) mode. The valid values are 0 and 1.

Figure 10-2 Regular Flip-Flop and Retention Flip-Flop

<pre>ff("IQ, IQZ") { next_state: "(D)"; clocked_on: "CLK";</pre>	<pre>ff("IQ, IQZ") { next_state: "((D) & (SLEEP' & WAKE))"; clocked_on: "CLK";</pre>

Figure 10-3 Regular Latch and Retention Latch

<pre>latch ("IQ, IQZ") { data-in: "(D)"; enable: "CZ";</pre>	<pre>latch ("IQ, IQZ") { data-in: "(D)"; enable: "((CZ) & (SLEEP' & WAKE))";</pre>

If the SLEEP and WAKE pins are disabled when “0”, the functional descriptions of the normal flip-flop or latch and the retention register in active mode are equivalent.

In the generated netlist, the retention pins are connected to the disabled state, either “1” or “0”, so the design behaves as if in normal operation until you provide the commands to wire them.

$$\text{power density of subarea} = \frac{\text{total power}}{\text{subarea}} \times \frac{\text{total currents of subarea}}{\text{total currents of white box}}$$

Interaction With Other Synopsys Tools

The use of retention registers is supported throughout the Synopsys Galaxy Platform. Some of the tools that support retention registers are briefly described below:

- Formality

The formal equivalence checking between RTL and gate-level design uses the state registers as compare points. When power-gating synthesis is done on a design, the gate-level design gets an extra state element, the retention latch, with every flip-flop, compared to the RTL design. Formality supports the attributes on the retention registers and therefore does not flag the retention latch as a violation.

- PrimeTime

The retention cells have power-gating pins. Such pins may be a function of the clock and therefore the cells in the library description may have timing arcs between clock pins and the power-gating pins. PrimeTime supports such timing arcs in a power-gating library cell.

11

Multivoltage Design Concepts

This chapter includes the following sections:

- [Multivoltage and Multisupply Designs](#)
- [Preparing Libraries for Multivoltage Designs With Shut-Down Blocks](#)
- [Using Target Library Subsets](#)
- [Power Compiler Flows for Multivoltage Designs](#)

Multivoltage and Multisupply Designs

The logic synthesis tools support the following types of low power designs:

- Multivoltage
- Multisupply
- Mixed multivoltage and multisupply

For multivoltage designs, the subdesign instances (blocks) operate at different voltages. For multisupply designs, the block voltages are the same but multiple power supplies are used. In the mixed design case, some blocks operate at different voltages and other blocks operate at a common voltage.

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).

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 step the voltage up or down from the input side of the cell to the output side, and power domain isolation is handled by isolation cells, which do not step the voltage. Note, however, that an enable-type level shifter can be used as isolation cells.

Power Domains

Specifically, in logic synthesis, a power domain is defined as a logic grouping of one or more hierarchical blocks in a design that share the following properties:

- 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
- 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

Power domains can be defined in the RTL design by using the Verilog system task `$power` or the VHDL power procedure. The logic synthesis tool infers (`infer_power_domain` command) the power domains from this RTL code. Alternatively, you can directly define power domains by using the `create_power_domain` command. Top-down and bottom-up compile flows, as well as Automated Chip Synthesis, support power domains.

Note:

Power Domains and Voltage Areas

Corresponding to the power domains of logic synthesis, you define *voltage areas* in physical synthesis as placement areas for the cells of the power domains. Except for level shifter cells, all cells in a voltage area operate at the same voltage.

There must be an exact one-to-one relationship between logical power domains and physical voltage areas. Design Compiler and IC Compiler can align the logic hierarchies of the power domains with their voltage areas with appropriate specifications. *The power domain name and the voltage area name should be identical.*

If you do not make these specifications, you are responsible for ensuring that the logic hierarchies are correctly aligned, as well as being correctly associated with the appropriate operating conditions.

Level Shifter and Isolation Cells

Level shifter cells are required to connect drive and load pins operating at different voltages across the power domains. These cells are modeled either as simple buffers or as buffer cells with an enable pin. Level shifter cells either step up or step down the voltage from their input side to their output side.

Generally, enable-type level shifters are used when the power domains of a design must be powered on and off independently, as well as stepping the voltage up or down. Designs can contain both buffer-type and enable-type level shifters.

Isolation cells are used when a power domain must be isolated from other power domains but the voltage is not required to change from the input side to the output side of the cell.

Note:

An enable-type level shifter can be used to connect drive and load pins across power domains operating at the same voltage (that is, the voltage is not stepped). In this application, the enable-type level shifter is equivalent to an isolation cell, and it can be used in place of isolation cell. The tool can make such a replacement during compile.

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 can be inserted either automatically as part of compilation or manually by using specific commands that insert level shifters. Similarly Isolation cells and enable-type level shifters can be instantiated at the RTL level of the design description or inserted manually by using commands that insert isolation cells.

The following level-shifter and isolation cell requirements should be observed:

- Two power supplies are usually required.
- Only buffer-type or enable-type level-shifter library cells are supported. Recall that an enable-type level shifter is basically a buffer-type level shifter but with an additional enable pin.
- Buffer-type and enable-type level-shifter library cells must have the `is_level_shifter` library attribute set to `true`.
- Enable-type level shifters must also have the `level_shifter_enable_pin` library attribute set on the enable pin.
- Isolation library cells must have the `is_isolation_cell` library attribute set to `true`.
- Isolation cells must have the `isolation_cell_enable_pin` library attribute set on the enable pin.
- Level shifters and isolation cells are selected by the logic synthesis tool from the target libraries. Therefore, at least one of the libraries must contain the required cells.

Basic Library Requirements for Multivoltage Designs

The libraries must conform to the Liberty open library rules. Design Compiler, Automated Chip Synthesis, and IC Compiler, as well as Astro and JupiterXT, support multivoltage, multiple nonlinear delay model (NLDM) libraries characterized at different voltages.

Target library cells are selected on the basis of matching operating conditions between library cells and voltage area. You can further restrict the selection of these cells by using the `set_target_library_subset` command.

k-factors are not supported for multivoltage designs and are ignored if present in the libraries.

Preparing Libraries for Multivoltage Designs With Shut-Down Blocks

As already noted, multivoltage designs typically have some power domains that are shut down and powered up during the operation of the chip while other power domains remain powered up. The always-on paths starting from an always-on block must connect to the specific pins of always-on cells in the power-down block. These cells can be special, dual power cells (isolation cells, enable-type level shifters, retention registers, special RAMs, and so on) or standard cells that when placed are confined to special always-on site rows within the power-down block.

The rules for marking a pin with the always-on attribute are as follows:

- Any input pin of a library cell can be marked always-on
- Any input pin of the instance can be marked always-on
- Either input or output pin of the hierarchical cell can be marked always-on

Specific commands are supported by the tool can be used to specify the always-on methodology to be applied to a particular power-down block. If special cells are used, they need to be marked appropriately so that the tool can determine the always-on paths and correctly optimize these paths.

Marking Always-on Library Cells

Only buffers and inverters can be used as dual-power, always-on cells. They must have two rails connections: a primary rail that is connected to a shut-down power supply, and a secondary rail that is connected to an always-on power supply. Because there is currently no permanent library attribute that identifies always-on cells, you must manually identify them in each time you run the tool by using the `set_attribute` command.

For example, to mark the retention register `RETENTION_A01` as an always-on cell, run the following command as part of a Design Compiler script:

```
set_attribute (RETENTION_A01) always_on true
```

Note, however, that marking certain library cells as always-on cells means that they are used only for always-on paths.

In cases where you intend to use a cell in both power-down and non-power-down blocks, you should not mark the library cell as always-on. Instead, you mark particular instances of the library cell as always-on.

Marking Always-on Library Pins

In addition to marking a library cell as always-on, you must also mark the always-on pin of the cell on a per session basis. For example, to mark the enable pin of the library cell `A`, run the following command as part of a Design Compiler script:

```
set_attribute (A/en) always_on true
```

By default, the tool assumes the enable pin of the isolation cells and the control pin of multithreshold-CMOS switch cells are always-on. No special marking is required for these pins.

If you are marking only certain cell instances of a library cell as always-on, then you must also mark the always-on pins on those cell instances. (See the following [“Marking Leaf Cell Instance Pins as Always-on.”](#))

Marking Pass-Gate Library Pins

In the current implementation, the tool has the ability to stop always-on cells from connecting to cells with pass gate inputs. An always-on buffer should not drive a gate that has pass transistors at the inputs (pass-gate). Pass-gate input cells should be driven by a standard cell in a shut-down power domain. Therefore, if your library contains any of these cells, you must mark them as pass-gates in each session.

For example, to mark the pin A of the mux cell MUX1, run the following command as part of a Design Compiler script:

```
set_attribute [get_lib_pins lib_name/MUX1/A] pass_gate true
```

Marking Leaf Cell Instance Pins as Always-on

You can also mark the pins of leaf cell instances of the dual-power cells with the always-on attribute.

For example, to mark pin A of the leaf cell U1/U2/A, run the following command as part of the logic synthesis script:

```
set_attribute [get_pins U1/U2/A] always_on true
```

By specifically marking the instance pin as always-on, you can use a certain library cell pin as an always-on pin in one instance but not in another instance of the design.

Using Target Library Subsets

You can use the `set_target_library_subset` command to restrict the target library cells eligible for mapping the hierarchical cells of a block.

The command syntax is:

```
set_target_library_subset {library_list} -object_list {cell_list} [-top]
```

where:

- *library_list* is a list of target library file names, all of which must also be listed in the main library variable
- *cell_list* is a list of hierarchical cells (blocks or top level) for which the target library subset is used
- *-top* is included if you want to apply this command to the top-level design

Using this command on a hierarchical cell or at the top level applies the library subset constraint on all lower cells in the hierarchy, except for those cells that have a different library subset constraint explicitly set on them.

Note:

The `set_target_library_subset` command does *not* require a uniquified design before using the command. However, if you use the `insert_level_shifters` command in the Design Compiler flow, you must first run `uniquify`.

To remove a target library subset constraint, use the `remove_target_library_subset` command with the appropriate library list and cell list.

You can check for errors and conflicts introduced through target library subsets by using the `check_mv_design` command with the `-target_library_subset` option. This command performs the following checks:

- Conflicts between target library subsets and the global `target_library` variable
- Conflicts between operating condition and target library subset
- Conflicts between the library cell of a mapped cell and target library subset

Use the `report_target_library_subset` command with the appropriate library list and cell list to find out what target library subsets have been defined for which hierarchical cells or for the top level.

Power Compiler Flows for Multivoltage Designs

Power Compiler power optimization supports multivoltage designs as well as single voltage designs. In particular, this includes the following power optimization flows:

- Clock-gating flow
- Power-gating flow
- Gate-level power optimization flow

You can use the `report_clock_gating` command with the `-gating` option to check the results of these flows.

Clock-Gating Flow

The steps for the clock-gating flow are:

1. Set target libraries
2. Set clock-gating style
3. Set variables
4. Read RTL design
5. Read constraints

6. Set operating conditions on blocks
7. Insert clock gates
8. Compile (`compile_ultra` command is recommended)

If clock gating includes test logic, then use the `insert_dft` command to hookup all the integrated clock-gate test pins to the top-level. This command automatically inserts level shifters on multivoltage designs.

Power-Gating Flow

The steps for the power-gating flow are

1. Set target libraries
2. Enable power-gating variable
3. Set other variables
4. Read RTL design
5. Read constraints
6. Set operating conditions on blocks
7. Set power-gating style
8. Compile (`compile_ultra` command is recommended)
9. Use power-gating signal specification for correct hookup
10. Hookup power-gating ports
11. Check and insert additional level shifters as needed

In this flow, the `insert_level_shifters` command is required after you run the `hookup_power_gating_ports` command.

Gate-Level Power Optimization Flow

Power Compiler supports all dynamic and leakage optimization flows for multivoltage designs.

The steps for a one-pass leakage optimization flow are

1. Set target to all multivoltage threshold libraries
2. Read RTL designs
3. Set optimization constraints
4. Set operating conditions on blocks

5. Check pre-existing level shifters and remove or preserve as needed
6. Compile (`compile_ultra` command is recommended)

For more information about these power optimization flows see Chapters 7, 8, and 9.

12

IEEE 1801™ (UPF) Flow For Multivoltage Design Implementation

This chapter describes the multivoltage design concepts and using the IEEE 1801™ Unified Power Format (UPF) to synthesize your multivoltage designs in Power Compiler. How to specify your power intent in the UPF file, reading the UPF file in Power Compiler, and commands supported in UPF mode are discussed in detail. This chapter contains the following sections:

- [Multivoltage Design Concepts](#)
- [Basic Library Requirements for Multivoltage Designs](#)
- [Synthesizing Multivoltage Designs Using UPF](#)
- [Defining Power Domains and Supply Network in UPF](#)
- [Defining Power State Tables](#)
- [Multivoltage Power Constraints](#)
- [Defining Multivoltage Design Strategies](#)
- [Handling Always-On Logic](#)
- [Using Basic Gates as Isolation Cells](#)
- [Additional Commands to Support Multivoltage Designs](#)
- [Reporting Commands in the UPF Flow](#)

Multivoltage Design Concepts

In multivoltage design, the subdesign instances operate at different voltages. In multisupply designs, the voltages of the various subdesigns are the same, but the blocks can be powered on and off independently. (In this user guide, unless otherwise noted, the term *multivoltage* includes multisupply and mixed multisupply-multivoltage designs.)

Multivoltage designs and design flows typically make use of the following features:

- Power domain – a grouping of logic hierarchies comprising a block, operating under a shared power supply
- Voltage area – physical placement area for the cells of a power domain
- Special cells – level shifters and isolation cells used to connect the drive and load pins across different power domains

Power Domains

Multivoltage designs contain design partitions which have specific power behavior compared to the rest of the design. A power domain is a basic concept in the Synopsys low-power infrastructure, and it drives many important low-power features across the flow.

By definition, a power domain is a logical grouping of one or more logic hierarchies in a design that share the same power characteristics, including:

- Power net hookup requirements
- Power down control & acknowledge signals, if any
- Power switching style

Thus, a power domain describes a design partition, bounded within logic hierarchies, that has a specific power behavior with respect to the rest of the design.

Each power domain has a supply network consisting of supply nets and supply ports and may contain power switches. The supply network is used to specify the power and ground net connections for a power domain. A supply net is a conductor that carries a supply voltage or ground. A supply port is a power supply connection point between the inside and outside of the power domain. Supply ports serve as the connection points between supply nets. A supply net can carry a voltage supply from one supply port to another.

When used together, the power domain and supply network objects allow you to specify the power management intentions of the design.

Every power domain must have one primary power supply and one primary ground. In addition to the primary power and ground nets, a power domain can have any number of additional power supply and ground nets.

A power domain has the following characteristics:

- Name
- Level of hierarchy or scope where the power domain is defined or created
- The set of design elements that comprise the power domain
- Associated set of supply nets that are allowed to be used within the power domain
- Primary power supply and ground nets
- Synthesis strategies for isolation, level-shifters, always-on cells, and retention registers

Note:

A power domain is strictly a synthesis construct, not a netlist object. For more information about the concept of Power Domain, see Power Intent Specification chapter in the *Synopsys Low-Power Flow User Guide*.

Voltage Area

A voltage area is the physical implementation of a power domain. A voltage area is associated with a power domain in a unique, tightly bound, one-to-one relationship. A voltage area is the area in which the cells of specific logic hierarchies are to be placed. A single voltage area must correspond to another single power domain, and vice versa. The power domains of a design are defined first in the logical synthesis phase and then the voltage areas are created in the physical implementation phase, in Design Compiler Topographical or in IC Compiler. The information that pertains to logic hierarchies, which belongs to a voltage area boundary is derived from a corresponding power domain. Also, all the cells that belong to a given voltage area have the power behavior described by the power domain characteristics. For more information about voltage area, see Synopsys Low-Power Flow User Guide.

Level-Shifter and Isolation Cells

In a multivoltage design, level-shifter cells are required when signals cross one power domain into another. The basic function of a level shifter is to connect the power domains by stepping the voltage up or down as needed.

In a design with power switching, an isolation cell is required where each logic signal crosses from a power domain that can be powered down to another power domain that is not powered down. The isolation cell operates as a buffer when the input and output sides

of the isolation cell are both powered up. When the input side of the cell is powered down, the isolation cell provides a constant output signal. An enable input controls the operating mode of the isolation cell.

A cell that can perform both level-shifting and isolation functions is called as an enable level-shifter cell. Design Compiler supports both simple buffer-type level shifters and enable-type level shifters, which are essentially buffer-type level shifters with an additional enable pin. Enable-type level shifters are used when the power domains must be powered on and off independently. A design can contain both types of level shifters.

Always-on Logic Cells

In multivoltage designs, certain cells inside a power domain need to remain powered-up even when the power domain is shut down. These cells are called the always-on logic cells. Multivoltage designs can also have power domains that have to be always-on while some power domains can be powered-down. In such cases, the control nets that connect cells in the always-on power domain to cells in the shut-down power domain must remain powered on when the power domain is shutdown. These paths are referred to as always-on paths.

Always-on cells can be of the following two types:

- **Single Power Standard Cells**

The standard cells, buffers, and inverters in the target library can be used as always-on. For Power Compiler to use standard cells, buffers, and inverters as always-on paths, you must set the always-on strategy on the power domain. For more details on defining always-on strategy, see [“Handling Always-On Logic” on page 12-25](#).

- **Dual Power Special Cells**

Special dual power buffers and inverters in the target library are used by the tool during always-on synthesis. Power Compiler automatically infers the backup power supply for these cells based on the supply load on these cells. The backup power supply for an always-on cell is determined by the always-on anchor point. If the always-on anchor point is an isolation cell, the backup power supply of the isolation cell is used as the backup power supply for the always-on buffer or inverter. Note that for an always-on cell in another power domain, the backup power supply must be available in the power domain in which the buffers or inverters are being inserted.

For more details See [“Handling Always-On Logic” on page 12-25](#).

Retention Register Cells

In multivoltage designs, when a power domain is shut down and later powered up, it is often necessary for the power domain to resume operation based on its last good state. Special cells called retention registers can store the state during the shutdown. A retention register has control signals to save and restore data. These control signals are distributed globally to ensure valid data at the retention circuit for shut-down and power-up modes.

Retention cells occupy more area than regular flip-flops. These cells continue to consume power when the power domain is powered down.

Basic Library Requirements for Multivoltage Designs

To synthesize your multivoltage design using Power Compiler, the target libraries you use must conform to the Liberty open library rules. The target libraries should also support special cells such as clock-gating cells, level-shifters, isolation cells, retention registers, and always-on buffers and inverters. To support synthesis of multivoltage designs, the tool also supports multiple libraries characterized at different voltages.

Note:

k-factors are not supported for multivoltage designs and are ignored if present in the libraries.

Power and Ground Pin Syntax

In the traditional, non-multivoltage designs, all components of the designs are connected to a single power supply at all times. So, the technology libraries used for synthesizing such designs do not contain any details on power supply and ground because all the cells are connected to the same type of VDD and VSS.

For the synthesis of multivoltage designs, it is necessary to specify the power supplies that can be connected to specific power pins of the cells. In order to accommodate this type of information, the Liberty library syntax was expanded to support the specification of power rail connection to the power supply pins of the cells.

So the target library that you specify for the synthesis of your multivoltage design should comply with the power and ground (PG) pin Liberty library syntax. This PG pin information is used by Power Compiler during the synthesis process.

Target Library Subsetting

Target library cells are selected on the basis of matching operating conditions between library cells and the power domain. The selection of these cells can be further restricted by using the `set_target_library_subset` command. Use this command to restrict the target library cells eligible for optimizing the hierarchical cells of a block.

The command syntax is follows:

```
set_target_library_subset {library_list} -object_list {cell_list} -top
```

where

- *library_list* is a list of target library file names, all of which must also be listed in the `target_library` variable.
- *cell_list* is a list of hierarchical cells (blocks or top level) for which the target library subset is used.

To use this command at the top level, you must include the `-top` option.

Using this command on a hierarchical cell or on the top-level design enforces the library restriction on all lower cells in the hierarchy, except for those cells that have a different library subset constraint explicitly set on them.

To remove a target library subset constraint, use the `remove_target_library_subset` command with the appropriate library list and cell list.

To check for errors and conflicts introduced by target library subsetting, use either the `check_mv_design -target_library_subset` command or the `check_target_library_subset` command. Both these commands check and report the following types of inconsistencies:

- Conflicts between target library subsets and the global `target_library` variable
- Conflicts between the operating condition of the current hierarchical block and the PVT values of the target library subset
- Conflicts between the library cell of a mapped cell and target library subset

Use the `report_target_library_subset` command with the appropriate library cell list to find the target library subsets that have been defined both for the hierarchical cells and at the top level.

Synthesizing Multivoltage Designs Using UPF

The Unified Power Format (UPF) is a standard set of Tcl-like commands used to specify the low-power design intent for electronic systems. UPF provides the ability to specify the power intent early in the design process. Also, UPF supports the entire design flow. For more information about the low-power flow and the various Synopsys tools that support UPF, see the Synopsys Low-Power Flow User Guide.

When you start the Design Compiler shell, by default the UPF mode is on. Use the `shell_is_in_upf_mode` command to check the mode of the Design Compiler shell. This command returns 1 when the Design Compiler shell is in UPF mode.

To start Design Compiler in non-UPF mode, use the `-non_upf_mode` option of the `dc_shell`. For more details on synthesizing your designs in non-UPF mode, see [Appendix C, “Implementing Multivoltage Designs Using RTL Isolation and Power Constructs.”](#)

Multivoltage Design Flow Using UPF

To synthesize your multivoltage design, the recommended method is to use the top-down approach. With your power intent defined in the UPF file, follow these steps to synthesize your multivoltage design:

1. Read your RTL file.

Use the `analyze` and `elaborate` commands to read the RTL source file. Use the `-format` option to specify the Verilog, SystemVerilog or VHDL file format.

You can also read an elaborated design using the `read_ddc` command. To get best results, read the design elaborated using the latest version of the tool.

2. Read the power definitions for your multivoltage design using the `load_upf` command.

In the UPF flow, the RTL file cannot have power definitions. Power Compiler issues an error message if it encounters power definitions in the RTL file. All the power definitions must be specified in the UPF file. This file can be used for synthesis, simulation, equivalence checking, and sign off.

By default, the `load_upf` command executes the commands in the associated UPF file, in the current level of hierarchy. The command errors out if the identifiers do not adhere to the naming rules specified in the UPF standard.

If you have modified the UPF file after reading it, you can use the `remove_upf` command to remove the UPF file. If you use the `remove_upf` command after compiling the entire design or a few subdesigns, the tool issues an error message. The tool also issues an error message if you use the `remove_upf` command after reading a compiled design.

To reload the UPF file after removing and updating it, use the `load_upf` command.

Note:

The Design Vision GUI provides a new dialog box, which you can access from the Power menu. Using the Generate UPF dialog box, you can generate a UPF script to set up the power domains, define their supply network, specify connections with other power domains, and define relationships with elements in the design hierarchy.

For more details about the Generate UPF dialog box, see the *Design Vision User Guide* and the Design Vision Online Help.

3. Specify the set of target libraries to be used.

Your target library must comply with the power and ground pin Liberty library syntax. The target library should also support special cells such as isolation cells and retention registers.

For more details on the target library requirement for multivoltage implementation see [“Basic Library Requirements for Multivoltage Designs” on page 12-5](#). For additional information about the PG pin Liberty library syntax, see the *Advanced Low Power Modeling* chapter in the Library Compiler User Guide.

4. Use the `set_operating_condition` command to set the operating condition on the top level of the design hierarchy and to derive the process and temperature conditions for the design. Use the `set_voltage` command to set the current operating voltage values for the power and the ground supply nets. For more details see [“Multivoltage Power Constraints” on page 12-17](#).

5. Specify your power optimization requirements.

Use the `read_saif` command to read the SAIF file containing the switching activity information. If you do not specify the toggle rate, a default value of 0.1 is used for propagating the switching activity.

Use the `set_max_leakage_power` or `set_max_dynamic_power` commands to optimize your design for leakage and dynamic power respectively.

When you use any of these power optimization constraints in the Design Compiler topographical technology, the tool also enables power prediction using the clock tree estimation. For more details about power prediction, see [“Performing Power Correlation” on page 6-10](#).

6. Compile your multivoltage design using the `compile` or `compile_ultra` command.

Use the `-gate_clock` option to insert the clock-gating logic during optimization.

Note:

If you are compiling your design for the first time and you are using the Design Compiler Topographical technology, it is recommended that you use the `compile_ultra -check_only` command. This option checks that your design and the libraries have all the data that is required by the `compile_ultra` command to successfully synthesize your design.

7. You can use the `check_mv_design -verbose` to check for multivoltage violations in your design.

This command performs several checks on your design such as inconsistencies in your design and the target libraries, violations related to level shifters, always-on cells, and so on.

Use the `-verbose` option to get the details of the violations. The `-max_messages` option controls the number of violations being reported. For more details, see the command man page.

8. Write the synthesized design using the `write -format` command. If you are writing the design in the ASCII format, it is recommended that you use the `change_names` command before you write out the design.

The write command by default writes the design in the .ddc (Synopsys logical database format) binary file format. You can write the design in verilog and VHDL (ASCII) formats. This can be used in subsequent Design Compiler sessions. In the Design Compiler topographical mode, you can use the `write_milkyway` and `write_parasitics` commands to write the synthesized design in the Milkyway and SPEF formats. These can be used in the IC Compiler flow.

To write the design constraints, use the `write_sdc` or the `write_sdf` command.

To generate the multivoltage reports, use the various reporting commands such as `report_power_domain`. For more details on multivoltage reporting commands, see [“Reporting Commands in the UPF Flow” on page 12-29](#).

9. Use the `save_upf` command to save the updated power constraints in another UPF file.

The UPF file written by Design Compiler, after the completion of the synthesis process, is used as input to the downstream tools, such as IC Compiler, PrimeTime or PrimeTime PX, and Formality. This file is similar to the one read into Design Compiler, but with the following additions:

- An additional comment on the first line of the UPF file generated by Design Compiler. An example is as follows:

```
#Generated by Design Compiler(B-2008.09) on Thu Aug 7 14:26:58 2008
```

- Explicit power connections to special cells such as level-shifter cells and dual supply cells.
- Any additional UPF commands that were specified at the command prompt in the Design Compiler session.

If you have specified UPF commands at the Design Compiler command prompt during synthesis, update the UPF file along with your RTL design with these commands. Without this update to the UPF file, Formality will not be able to successfully verify the design.

Note:

You can synthesize your multivoltage design without using the UPF flow. For information about multivoltage design implementation using the non-UPF flow, see [Appendix C, “Implementing Multivoltage Designs Using RTL Isolation and Power Constructs”](#).

Defining Power Domains and Supply Network in UPF

Hierarchy and Scope

The scope of the power domain is the logical hierarchy where the power domain is created. Design elements that belong to a power domain are said to be in the extent of the power domain. For more information, see the chapter Power Intent Specification in the *Synopsys Low-Power Flow User Guide*.

Use the `set_scope` command to specify the scope or level of hierarchy. The `set_scope` command sets the scope or the level of hierarchy to the specified scope. When no instance is specified, the scope is set to the top level of the design hierarchy. Alternatively, you can also use the `current_instance` command to specify the current scope. However, in the power context, the `set_scope` command is preferred.

You should explicitly specify the scope using the `set_scope` or the `current_instance` command. Unless explicitly specified, Design Compiler uses the current scope or current level of hierarchy when you define objects. For more information about scope, see the Power Intent Specification chapter in the *Synopsys Low-Power Flow User Guide*.

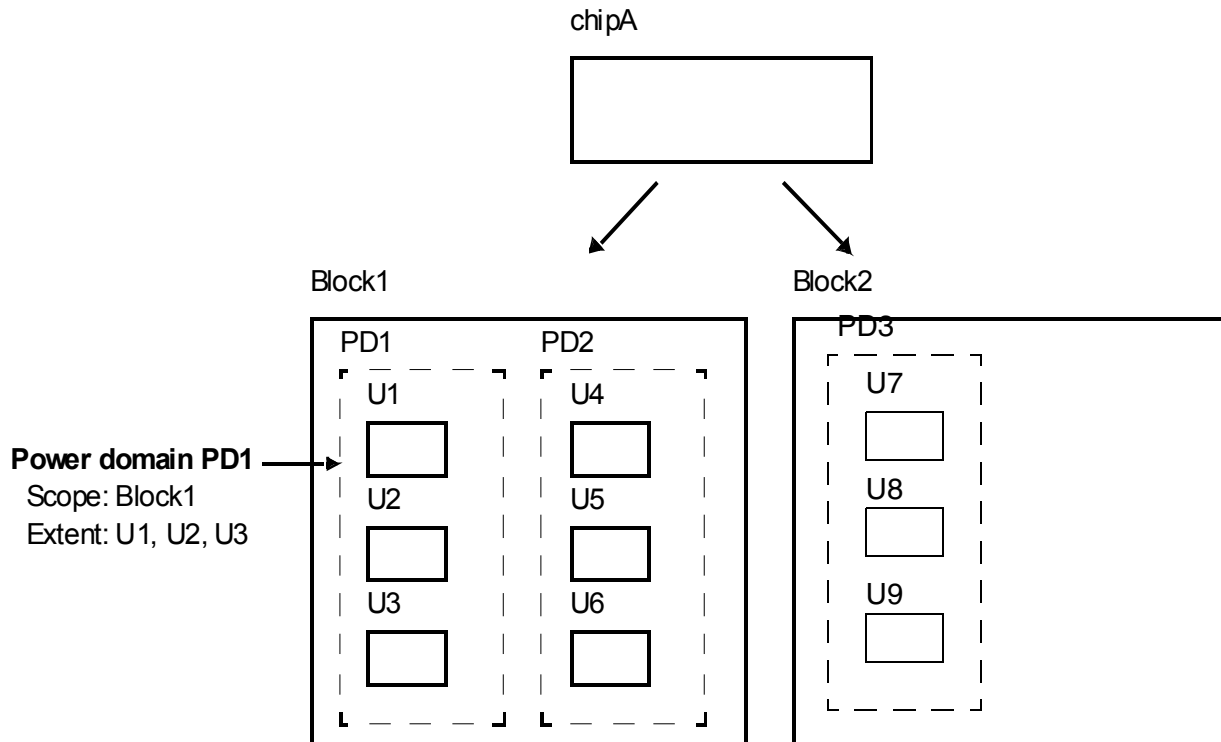
Creating Power Domains

Use the `create_power_domain` command to create a power domain with the specified name. The syntax of the `create_power_domain` command is as follows:

```
create_power_domain [-elements element_list] [-include_scope] [-scope  
instance_name] domain_name
```

Use the `-element` option to specify the list of hierarchical, I/O or pad cells that are added as extent of the power domain. The `-include_scope` option specifies that all the elements in the current scope share the primary supply of the power domain, but are not necessarily added to the extent of the power domain. Use the `-scope` option to specify the logical hierarchy or the scope at which the power domain is to be defined. *domain_name* is the name of the power domain to be created.

Figure 12-1 Defining a Power Domain and scope



The following example creates power domains PD1 and PD2, as specified in [Figure 12-1](#):

```
create_power_domain -elements {U1 U2 U3} -scope Block1 PD1
create_power_domain -elements {U4 U5 U6} -scope Block1 PD2
```

Alternatively, you can use the `set_scope` command to first set to the desired scope and then to create the power domain, as mentioned in the following example:

```
set_scope Block1
create_power_domain -elements {U1 U2 U3} PD1
create_power_domain -elements {U4 U5 U6} PD2
```

You can use the `-include_scope` option to include all the elements in the specified scope to share the supply of the power domain. However the elements in the specified scope are not necessarily added to the power domain.

```
create_power_domain -elements {U7 U8} -include_scope Block2 PD3
```

In this case, the element U9 shares the supply of power domain PD3, though U9 is not explicitly mentioned to be part of the power domain PD3.

Creating Supply Port

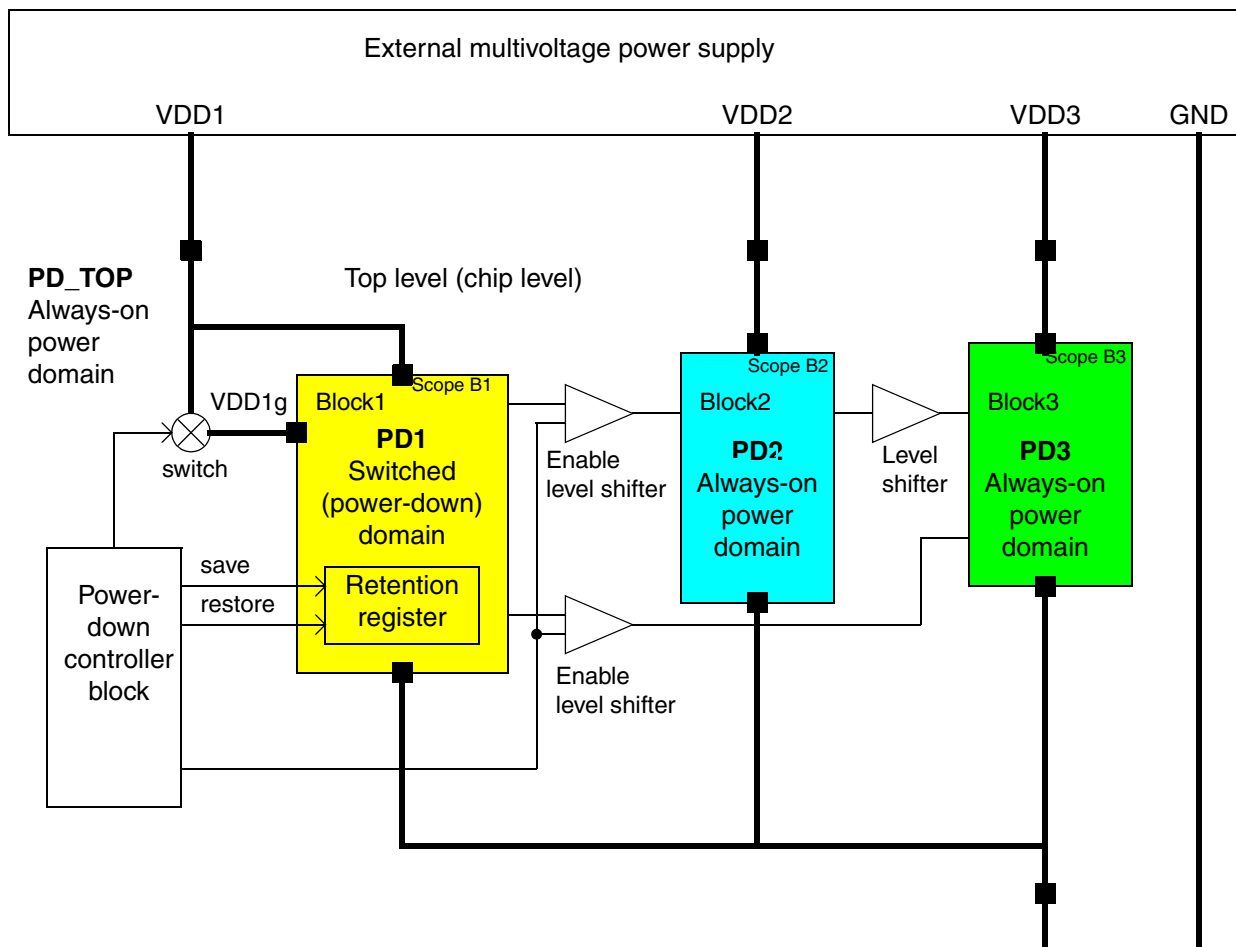
To create the supply and ground ports, use the `create_supply_port` command.

The syntax of the `create_supply_port` command is as follows:

```
create_supply_port supply_port_name [-domain power_domain_name ]
[-direction in|out]
```

The `supply_port_name` specified should be unique at the level of hierarchy it is defined. The name of the supply port should be a simple (non-hierarchical) name. Unless the `-domain` option is specified, the port is created in the current scope or level of hierarchy and all power domains in the current scope can use the created port. By default the direction of the port is in or input port.

Figure 12-2 Power Intent Specification



The following example shows the creation of the ports shown in [Figure 12-2](#):

To create the supply ports VDD1, VDD2 and VDD3 and GND at the top level of design hierarchy or power domain PD_TOP use the command as follows:

```
create_supply_port VDD1
create_supply_port VDD2
create_supply_port VDD3
create_supply_port GND
```

To create the supply ports VDD1, VDD1g and GND in the power domain PD1, use the `create_supply_port` command as follows:

```
create_supply_port VDD1 -domain PD1
create_supply_port VDD1g -domain PD1
create_supply_port GND -domain PD1
```

To create the supply ports VDD2, GND in the power domain PD2, and VDD3, GND in power domain PD3 respectively, use the `create_supply_port` command as follows:

```
create_supply_port VDD2 -domain PD2
create_supply_port GND -domain PD2
create_supply_port VDD3 -domain PD3
create_supply_port GND -domain PD3
```

Note:

Connectivity is not defined when the supply port is created. To define connectivity use the `connect_supply_net` command.

Creating Supply Nets

A supply net connects supply ports and or supply pins. Use the `create_supply_net` command to create a supply net. The syntax of the `create_supply_net` command is as follows:

```
create_supply_net -domain domain_name [-reuse] supply_net_name
```

The supply net is created in the same scope or logical hierarchy as the specified power domain. When the `-reuse` option is used, the specified supply net is not created; instead an existing supply net with the specified name is reused.

```
create_supply_net GND_NET
create_supply_net -domain PD1 GND_NET
create_supply_net -domain PD2 GND_NET
create_supply_net -domain PD3 GND_NET
```

When a supply net is created it is not considered a primary power supply or ground net. To make a specific power supply or ground net of a power domain, as primary supply or ground net, use the `set_domain_supply_net` command.

Connecting Supply Nets

This command connects the supply net to the specified supply ports or pins. The connection can be within the same level of hierarchy or to ports or pins down the hierarchy. The syntax of the `connect_supply_net` command is as follows:

```
connect_supply_net supply_net_name [-ports | -pins list]
```

The following example depicts the use of the `connect_supply_net` command to connect supply nets to various supply ports in different levels of hierarchy or power domains.

```
connect_supply_net GND_NET -ports GND
connect_supply_net GND_NET -ports {B1/GND B2/GND B3/GND} GND
```

Specifying Primary Supply for Power Domains

Use the `set_domain_supply_net` command to define the primary power supply net and primary ground net for a power domain. The syntax of the `set_domain_supply_net` command is as follows:

```
set_domain_supply_net -primary_power_net supply_net_name
-primary_ground_net supply_net_name domain_name
```

Every power domain must have one primary power and one ground connection. When a supply net is created it is not a primary supply net. You must use the `set_domain_supply_net` command to designate the specific supply net the primary supply net for the power domain. All cells in a power domain are assumed to be connected to the primary power and ground net of the power domain to which the cells belong. If the power or ground pins of a cell in a power domain, is not explicitly connected to any supply net, the power or ground pin of the cell is assumed to be connected to the primary power or ground net of the power domain to which the cell belongs.

When in the scope of Top, you can use the following command to designate VDD and GND nets as the primary power and ground net, respectively, of the power domain PD_TOP.

```
set_domain_supply_net -primary_power_net VDD -primary_ground_net GND
PD_TOP
```

Creating Power Switch

The `create_power_switch` command creates a virtual instance of power switch in the scope of the specified power domain. The power switch has at least one input supply port and one output supply port. When the switch is off, the output supply port is shut down and has no power. The syntax of the `create_power_switch` command is as follows:

The `create_power_switch` command is only checked for syntax in Power Compiler. This command lets the tool know that a generic power switch resides in the design at a specific scope or level of hierarchy. It does not use the power switch information for synthesis. It merely passes on power switch commands in the input UPF file to its output UPF file. The power switch is instantiated and used in IC Compiler.

Following is a simple power switch definition for the power switch in [Figure 12-2](#).

```
create_power_switch SW1 \
  -domain PD_TOP \
  -output_supply_port {SWOUT VDD1g} \
  -input_supply_port {SWIN1 VDD1} \
  -control_port {CTRL swctl} \
  -on_state {ON VDD1 {!swctl}}
```

Adding Port State Information to Supply Ports

The `add_port_state` command adds state information to a supply port. This command specifies the name of the supply port and the possible states of the port. The first state specified is the default state of the supply port. The port name can be a hierarchical name. Each state is specified as a state name and the voltage level for that state. The voltage level can be specified as a single nominal value, set of three values (minimum, nominal, and maximum), or 0.0, or the keyword off to indicate the off state. The state names are also used to define all possible operating states in the Power State Table. The syntax of the `add_port_state` command is as follows:

```
add_port_state port_name {-state {name nom| min nom max | off}} *
```

*Asterisk indicates an item that can be repeated in the command.

A power switch supply port is considered a supply port because it is connected by a supply net, so it can be specified as the supply port in the `add_port_state` command. Note that supply states specified at different supply ports are shared within a group of supply nets and supply ports directly connected together. However, this sharing does not happen across a power switch.

[Example 12-1](#) shows the definition of states for the power nets:

Example 12-1 Defining the States of the Power Nets

```
add_port_state header_sw/VDD \
  -state {HV 0.99} \
  -state {LV 0.792} \
  -state {OFF off}
```

[Example 12-2](#) shows the definition of states for the ground nets:

Example 12-2 Defining the States of the Ground Nets

```
add_port_state footer_sw/VSS \
  -state {LV 0.0} \
```

```
-state {OFF off}
```

Defining Power State Tables

A power state table (PST) defines the legal combination of states that can exist simultaneously during the operation of the design. A PST is a set of power states of a design in which each power state is represented as an assignment of power states to individual power nets. A PST of a design captures all the possible operational modes of the design in terms of power supply levels. Given a PST, a power state relationship (including voltage and relative always-on relations) can be inferred between any two power nets. The PST is used by the synthesis tool for analysis, synthesis, and optimization of the multivoltage design.

Creating Power State Table

The `create_pst` command creates a new power state table and assigns a name to the table. The command lists the supply ports or supply nets in a particular order. The `add_port_state` defines the name of the possible states for each supply port.

The power switch supply ports are considered supply ports because they are connected by supply nets, so they can be listed as supply nets in `create_pst` command. A supply port and a supply net can have the same name, even when they are unconnected. If such a name is listed in the `create_pst` command, it is assumed to represent the supply port and not the supply net. The syntax of the `create_pst` command is as follows:

```
create_pst table_name -supplies list
```

Defining the States of Supply Nets

The `add_pst_state` command defines the states of each of the supply nets for one possible state of the design. The command must specify the name of the state, the name of the power state table previously created by the `create_pst` command, and the states of the supply ports in the same order as listed in the `create_pst` command.

The listed states must match the supply ports or nets listed in the `create_pst` command in the corresponding order. For a group of supply ports and supply nets directly connected together, the allowable supply states are derived from the shared pool of supply states commonly owned by the members of the group.

The following example creates a power state table, defines the states of the supply ports, and lists the allowed power states for the design.

```
create_pst pt -supplies { PN1 PN2 SOC/OTC/PN3, FSW/PN4 }
add_port_state PN1 -state { s88 0.88 }
add_port_state PN2 -state { s88 0.88 } -state { s99 0.99 }
add_port_state SOC/OTC/PN3 -state { s88 0.88 } -state { pdown off }
add_port_state FSW/PN4 -state { s0, 0.0 } -state { pdown off }
add_pst_state s1 -pst pt -state { s88 s88 s88 s0 }
```

```
add_pst_state s2 -pst pt -state { s88 s88 pdown s0 }  
add_pst_state s3 -pst pt -state { s88 s99 pdown s0 }  
add_pst_state s4 -pst pt -state { s88 s99 s88 pdown }
```

Multivoltage Power Constraints

Use the `set_operating_condition` command to set the operating conditions for the top level of your multivoltage design. To specify the operating voltage for the supply nets in the design, use the `set_voltage` command. All supply nets including the ground, need to be assigned a voltage value. If any of the supply nets have not been assigned voltages, Power Compiler errors out during the execution of the `compile_ultra` or the `compile` command. Before compiling your design, you can use the `check_mv_design -power` to check that you have defined the operating voltages for all the supply nets.

UPF constraints are mapped during the execution of `compile` or `compile_ultra` command. However following are the two exceptions:

- Power switches are not mapped during the compilation process because Design Compiler does not use power switches. Power Compiler passes on power switch commands in the input UPF file to its output UPF file.
- Mapping of retention registers takes place during the `compile` or `compile_ultra` command. However, if the `map_retention_cell` command is not used before compiling the design, the `compile` or `compile_ultra` command does not map the sequential element to a retention register.

Defining Multivoltage Design Strategies

To make the best use of multivoltage implementation techniques, the design requires fundamental enhancements, such as the use of special cells to handle the voltage differences and multiple power supplies. In UPF, insertion of the special cells is based on strategies that you define for each type of special cell.

UPF supports

- Commands to specify the strategy for inserting the special cells in power domains operating at different voltages
- Power domains that can be powered down
- Special cells in the power-down domains that can restore their previous state when the power domain is turned on

When defining strategies on power domains, the strategy names have to be unique in the scope or the level of hierarchy in which the strategies are defined.

Name Format

When Power Compiler inserts a special cell in the design, it assigns an instance name to the new cell by adding a prefix or a suffix to the name of the object that is being isolated or level-shifted - that can be a port, pin, or net. You use this command to specify the prefix or suffix to be used for the level-shifter cells and the isolation cells. The `name_format` command uses the following syntax:

```
name_format
  [-isolation_prefix string]
  [-isolation_suffix string]
  [-level_shift_prefix string]
  [-level_shift_suffix string]
```

If the name generated conflicts with another previously defined name in the same name space, the generated name is further extended by an underscore character followed by a positive integer. An empty string is a valid value for any prefix or suffix option. When the prefix and suffix are both an empty string, only the underscore and the number string combination are used as a suffix to create the new instance name for the isolation or level-shifter cell.

The following section describes how you specify the strategies for the various types of special cells that are required in multivoltage design implementation:

- [Defining the Level-Shifter Strategy](#)
- [Defining the Isolation Strategy](#)
- [Defining the Retention Strategy](#)

Defining the Level-Shifter Strategy

The level-shifter commands let you specify the strategy for inserting level-shifter cells between power domains that operate at different voltages. Level shifters are also inserted automatically by the tool during execution of the `compile` or `compile_ultra` commands.

You use the `set_level_shifter` command to specify a strategy for inserting level shifters during the `compile_ultra` command. Power Compiler inserts level shifters on signals that have sources and sinks that operate at different voltages, following the specified strategy. If a level-shifter strategy is not specified for a particular power domain, the default level-shifter strategy applies to all elements in the power domain. The syntax of the `set_level_shifter` command is as follows:

```
set_level_shifter strategy_name
  -domain domain_name
  [-elements port_pin_list]
  [-applies_to inputs | outputs | both]
  [-threshold float]
```

```
[-rule low_to_high | high_to_low | both]
[-location self | parent | fanout | automatic]
[-no_shift]
```

The `-elements` option specifies a list of ports and pins in the domain to which the strategy applies, overriding any `-threshold` or `-rule` settings.

The `-threshold` option defines how large the voltage difference must be between the driver and sink before level shifters are inserted, overriding any such specification in the cell library. The `-rule` option can be set to `low_to_high`, `high_to_low` or `both`. If `low_to_high` is specified, signals going from a lower voltage to a higher voltage get a level shifter when the voltage difference exceeds the `-threshold` value. Similarly if `high_to_low` is specified, signals going from higher voltage to lower voltage get a level shifter when the voltage difference exceeds the `-threshold` value. The default behavior is `both`, which means that a level shifter is inserted in either situation.

The `-location` option specifies where the level-shifter cells are placed in the logic hierarchy:

- `self` - The level-shifter cell is placed inside the cell being level-shifted.
- `parent` - The level-shifter cell is placed in the parent of the cell being level-shifted.
- `fanout` - Level shifters are placed at all fanout locations (sinks) of the port being shifted.
- `automatic` - Power Compiler is free to choose the appropriate location. This is the default behavior.

Specifying a strategy does not force a level shifter to be inserted unconditionally. Power Compiler relies on the power state table and the specified rules, such as threshold, to determine where level shifters are needed. The tool issues a warning if it determines that a level shifter is not required.

The following strategies have decreasing order of precedence, irrespective of the order in which they are executed:

```
set_level_shifter -domain -elements (with optional -applies_to)
set_level_shifter -domain -applies_to <input/output>
set_level_shifter -domain (with optional -applies_to both)
```

It is an error to specify a strategy of the same precedence level explicitly on the same power domain or design elements as the previous strategy specification.

By default, Power Compiler does not insert level-shifter cells on clock nets. For the tool to insert level-shifter cells on specific clock nets, set the `auto_insert_level_shifters_on_clocks` variable to specific clock nets. You use the following variable setting to allow the tool to insert level-shifter cells on all clock nets that need level shifters.

```
set auto_insert_level_shifters_on_clock "all"
```

Associating Specific Library Cells With the Level-Shifter Strategy

When you specify the level-shifter strategy for a power domain, by default the tool maps the level-shifter cells to any suitable level-shifter cells in the technology library. You use the `map_level_shifter_cell` command to associate a specific set of library cells to the specified level-shifter strategy. This command does not force the insertion of the level-shifter cells. Instead, when the tool inserts the level-shifter cell, it chooses the library cells that are specified with the `-lib_cells` argument of the `map_level_shifter_cell` command. This command has no effect on instantiated level-shifter cells that have `dont_touch` attribute set on them. For more details, see the command man page.

Defining the Isolation Strategy

You use the `set_isolation` command to define the isolation strategy for a power domain and the elements in the power domain on which the strategy is applied. The syntax of the `set_isolation` command is as follows:

```
set_isolation isolation_strategy_name
  -domain power_domain
  [-isolation_power_net isolation_power_net]
  [-isolation_ground_net isolation_ground_net]
  [-clamp_value 0 | 1 | latch]
  [-applies_to inputs | outputs | both]
  [-elements objects]
  [-no_isolation]
```

Definition of an isolation strategy contains specification of the enable signal net, the clamp value, and the location (inputs, outputs, or both). At least one of `-isolation_power_net` or `-isolation_ground_net` options must be specified unless `-no_isolation` option is used. If only `-isolation_power_net` option is specified, the primary ground net is used as the isolation ground supply. If only `-isolation_ground_net` option is specified, the primary supply net is used as the isolation power supply. If both are used, they specify the supply nets to use as the isolation power and ground nets. The isolation power and ground nets are automatically connected to the implicit isolation circuit. If `-no_isolation` option is specified, it means that the elements in the `-elements` list will not be isolated.

The `-elements` option can be used to specify the elements to isolate in cases where there are multiple isolation strategies within a given power domain. The listed elements (input or output ports on the domain boundary) must be within the specified power domain. If `-elements` directly specifies a port by name (not implicitly, by specifying the port's instance or an ancestor of that instance), then the isolation strategy applies to that port regardless of whether that port's mode matches the one specified by the `-applies_to` option. Without the `-elements` option, the isolation strategy applies to the whole power domain.

Ports of a power domain refer to the logical ports of the root cells of the power domain, or in case of a power domain containing the top-level design, logical ports of the design. Input ports of a power domain are the ports defined as inputs in the corresponding HDL module. Similarly, output ports of a power domain are the ports defined as outputs in the corresponding HDL module.

The `-clamp_value` option specifies the constant value in the isolation output: 0, 1, latch. The latch setting causes the value of the non-isolated port to be latched when the isolation signal becomes active.

Note:

Power Compiler does not support the value `z` for the `-clamp_value` option. 0 and 1 are the only supported values.

The `-applies_to` option specifies the parts of the power domain that are isolated: inputs, outputs or both. The default is outputs.

Although the power state table can potentially reduce the number of isolation cells required, isolation synthesis is entirely based on directives set with the `set_isolation` and `set_isolation_control` commands.

Power Compiler can perform certain types of optimization on isolation circuits, as long as the functionality is not affected. For example, suppose you have two blocks, A and B, with signals going from A to B. You specify output isolation on A (in the parent) and input isolation on B (in the parent). If the strategy results in two back-to-back isolation cells with no fan out in between, Power Compiler can merge the isolation cells. The tool can merge the isolation cells based on the enable signal, power or ground signals.

Power Compiler requires that the isolation power and ground nets operate at the same voltage as the primary and ground nets of the power domain where the isolation cells will be located.

The following strategies have decreasing order of precedence, irrespective of the order in which they are executed:

```
set_isolation -domain -elements (with optional -applies_to)
set_isolation -domain -applies_to <input/output>
set_isolation -domain (with optional -applies_to both)
```

Every isolation strategy defined by a `set_isolation` command must have a corresponding `set_isolation_control` command, unless the strategy is `-no_isolation`.

The `set_isolation_control` command allows the specification of the isolation control signal and sense separately from the `set_isolation` command. The command identifies an existing isolation strategy and specifies the isolation control signal for that strategy. The syntax of the `set_isolation_control` command is as follows:

```
set_isolation_control isolation_strategy_name
    -domain power_domain
    -isolation_signal isolation_signal
[-isolation_sense 0 | 1]
[-location self | parent ]
```

The tool can identify isolation cells in the power domain across the design hierarchy and associate them with UPF strategies. To identify the isolation cells, the tool uses the location value you specify using the `-location` option of the `set_isolation_control` command. When the value you specify is `self`, the tool starts the search from the port on the boundary of the power domain and traverses inside the power domain until it encounters either a cell, a multiple fanout net, or the boundary of another power domain. When the location you specify is `parent`, the tool starts the search from the port on the boundary of the power domain and traverses outside the power domain until it encounters a cell, a multiple fanout net, or the boundary of another power domain.

If the cell encountered is an isolation cell that is not already associated with an isolation strategy, the tool associates the cell with an appropriate isolation strategy. This association is based on the values you specified with the `-clamp_value` option of the `set_isolation` command and the `-isolation_sense` option of the `set_isolation_control` command. If the cell encountered is not an isolation cell, the tool does not treat the port as an isolation port, and during the next optimization step, the tool inserts an isolation cell.

The `-isolation_sense` option specifies the logic state of the isolation control signal that places isolation cells in the isolation mode. The possible values for this option are 0 or 1. The default is 1. The isolation signal specified by the `-isolation_signal` option can be for a net or a pin or port, with the net having higher precedence. The isolation signal need not exist in the logical hierarchy where the isolation cells are to be inserted; the synthesis or implementation tool can perform port-punching as needed to make the connection. Port-punching means automatically creating a port to make a connection from one hierarchical level to the next. These punched ports are not considered for isolation or level-shifting, even though after the port creation, these ports reside within the coverage of an isolation or level-shifter strategy.

Existing ports are isolated and level-shifted according to the applicable isolation and level-shifter strategy, even if they reside on an always-on path, a logic path marked as always-on relative to the receiving end.

Associating Specific Library Cells With the Isolation Strategy

When you define an isolation strategy, by default the tool associates the isolation strategy with any suitable isolation cell in the technology library. Using the `map_isolation` command you can associate a specified set of library cells with the isolation strategy. The `map_isolation_cell` command can also be used to associate normal cells used as isolation cells and enable-level-shifter cells with the isolation strategy.

When designs contain instantiated isolation cells that are associated with an isolation strategy, the `map_isolation_cell` command remaps these library cells to the cells specified with the `-lib_cells` argument of the command. If the instantiated isolation cells have `dont_touch` attribute set on them, the command does not remap these cells. The command has no impact on the instantiated isolation cells that are not, or cannot be associated with an isolation strategy. For more details see the command man page.

Defining the Retention Strategy

The retention commands specify the strategy for inserting retention cells inside the power-down domains.

The `set_retention` command specifies which registers in the power-down domain are to be implemented as retention registers and identifies the save and restore signals for the retention functionality. Only the registers specified in the `-elements` option are given retention capabilities. The syntax of the `set_retention` command is as follows:

```
set_retention retention_strategy_name
    -domain power_domain
    -retention_power_net retention_power_net
    -retention_ground_net retention_ground_net
    [-elements objects]
```

The `-elements` option specifies the objects in the specified power domain to which the retention strategy applies. The objects can be hierarchical cells, leaf-level cells, HDL blocks, and nets. If a design element is specified, then all registers within the design element acquire the specified retention strategy. If a process is specified, then all registers inferred by the process acquire the specified retention strategy. If a register, signal, or variable is specified and that object is a sequential element, then the implied register acquires the specified retention strategy. Any specified register, signal, or variable that does not infer a sequential element is not affected by this command. If `-elements` option is not used, it is the same as listing the elements that define the power domain, in other words the retention strategy is applied to all the unmapped sequential cells in the specified power domain.

At least one of the `-retention_power_net` or `-retention_ground_net` must be specified. If only `-retention_power_net` is specified, the primary ground net is used as the retention ground supply. If only `-retention_ground_net` is specified, the primary supply net is used as the retention power supply. If both are used, they specify the supply nets to use as the retention power and ground nets. The retention power and ground nets are automatically connected to the implicit save and restore processes and shadow register.

The following strategies have decreasing order of precedence, irrespective of the order in which they are executed:

```
set_retention -domain -elements
set_retention -domain
```

The power and ground nets of the retention registers can operate at voltage levels different from the primary and ground supply voltage levels of the power domain where the retention cell is located.

Every retention strategy defined by a `set_retention` command must have a corresponding `set_retention_control` command. The `set_retention_control` command allows the specification of the retention control signal and sense separately from the `set_retention` command. The command identifies an existing retention strategy and specifies the save and restore signals and senses for that strategy. The syntax of the `set_retention_control` command is as follows:

```
set_retention_control retention_strategy_name
  -domain power_domain
  -save_signal { save_signal high | low }
  -restore_signal { restore_signal high | low }
```

The `-save_signal` setting specifies the existing net, port, or pin in the design used to save data into the bubble register prior to power-domain; and the logic state of the signal, either low or high, that causes this action to be taken.

Similarly, the `-restore_signal` setting specifies the existing net, port, or pin in the design used to restore data from the bubble register prior to power-up; and the logic state of the signal, either low or high, that causes this action to be taken.

Each control signal can be either a net or a pin or port, with net having higher precedence. The retention signal need not exist in the logical hierarchy where the retention cells are to be inserted. The synthesis or implementation tools perform port-punching, as needed, to make the connection. Port-punching means automatically creating a port to make a connection from one hierarchical level to the next. These punched ports are not considered for isolation or level-shifting, even though after the port creation, these ports reside within the coverage of an isolation or level-shifter strategy.

Mapping the Retention Registers

The `map_retention_cell` command provides a mechanism for constraining the implementation choices for retention registers. The command must specify the name of an existing retention strategy and power domain. The syntax of the `map_retention_cell` command is as follows:

```
map_retention_cell retention_strategy_name
  -domain power_domain
  [-lib_cells lib_cells]
  [-lib_cell_type lib_cell_type]
  [-elements objects]
```

The `-lib_cells` option specifies a list of target library cells to be used for retention mapping.

The `-lib_cell_type` option directs the tool to select a retention cell that has the specified cell type in the implementation model. Note that this option setting does not change the simulation semantics specified by the `set_retention` command.

The `-elements` option lists the register elements (directly or indirectly) within the domain to which the mapping command is applied. The elements must be included in the elements listed in the related `set_retention` command. If `-elements` is not used, all registers inferred from the retention strategy have the mapping applied.

The `retention_cell` attribute on the library cells in the target library defines the retention styles of the library cells.

Handling Always-On Logic

Multivoltage designs typically have power domains that are shut down and powered up during the operation of the chip while other power domains remain powered up. The control nets that connect cells in an always-on power domain to cells within the shut-down power domain must remain on during shutdown. These paths are referred to as always-on paths.

Marking Pass-Gate Library Pins

In its current implementation, the tool has the ability to prevent always-on cells from connecting to cells with pass-gate inputs. An always-on buffer should not drive a gate that has pass transistors at the inputs (pass-gate). Pass-gate input cells should be driven by a standard cell in a shut-down power domain. Therefore, if your library contains any of these cells, you must mark them as pass-gates in each session.

For example, to mark pin A of the multiplexer cell MUX1, run the following command:

```
dc_shell> set_attribute [get_lib_pins lib_name/MUX1/A] pass_gate true
```

Marking Leaf Cell Instance Pins As Always-On

You can also mark the pins of leaf cell instances of a library cell as always-on.

For example, to mark pin A of the leaf cell U1/U2/A, run the following command:

```
dc_shell> set_attribute [get_pins U1/U2/A] always_on true
```

By specifically marking the instance pin as always-on, you direct the tool to use this pin as an always-on anchor point. Note that for the proper always-on buffering to occur, the tool must be able to derive a backup power supply for the always-on buffers and inverters.

Marking Library Cells for Always-On Optimization

Design Compiler performs always-on buffering only when the target library contains an always-on inverter and an always-on buffer. To use a specific library cell in the optimization of always-on paths within the shut-down power domains, you mark the cell with the `always_on` attribute. The tool uses only always-on cells to optimize the always-on paths within the shut-down power domains. The cells that are not marked as always-on are used outside the shut-down power domains.

Note:

When you set the `always-on` attribute on a library cell, the tool does not use that library cell for optimization of other types of paths. If you want to use a library cell in both always-on paths and shut-down paths, you must set the `always-on` attribute only on the instances of the library cell that are present in the shut-down power domains.

Automatic Always-On Optimization

Power Compiler performs automatic constraining, marking and optimization of always-on nets, including the feedthrough nets, by default. The tool uses the related supply net of the load or the driver pin as the supply net for the inserted always-on buffers or inverters. The tool also ensures that no additional isolation or level-shifting violations are introduced by the automatic always-on synthesis.

To select the supply nets for the inserted buffers and inverters used in the always-on synthesis, the tool applies the following rules, in the specified order:

1. For a load net, when the related supply net of the load is in the same power domain as the net, the related supply net of the load is used.
2. For a driver net, when the related supply net of the driver is in the same power domain as the net, the related supply net of the driver is used.
3. For feedthrough nets with multiple choices of nets, related supply net of the load has precedence over the related supply net of the driver.

The tool marks the selected nets based on the following rules:

- When the related supply net is in the same power domain as the net and it is not the primary power net of the power domain, the tool marks the net as `always-on`.
- When the related supply net is not in the same power domain as the net, the tool marks the net as `dont_touch`.
- When the related supply net is in the same power domain as the net, and it is the primary power net of the power domain, the tool inserts a regular buffer or inverter, and the net is not specifically marked.

Performing Always-on Optimization on Top-level Feedthrough Nets

To perform always-on optimization on top-level feedthrough nets, you must follow these steps:

1. Set the `always-on` attribute on the output port that is driven by the feedthrough net.
2. Specify the related supply net information on the output port that is driven by the feedthrough net.

The tool derives the power and ground net information for the always-on buffering based on the related supply net you specify for the output port driven by the feedthrough net. If the tool detects a level-shifter violation or an isolation violation on a feedthrough net, it sets a `dont_touch` attribute on the feedthrough net. This is done to prevent the shifting of the violation from one power domain to another.

The following example script shows the sequence of commands used to enable always-on synthesis on top-level feedthrough nets.

```
set_attribute [get_ports out_a] always-on true
set_related_supply_net -power VDD_ON -ground VSS \
-object_list [get_ports out_a]
```

Identifying User-Defined Always-On Paths

The steps followed by the tool to identify user-defined always-on paths are slightly different from the steps it takes to identify automatically generated always-on paths. When the design objects are marked always-on by the tool, to identify always-on paths, the tool traverses back until it encounters the control signal that you specified through the UPF commands, such as the `set_isolation_control` or `set_retention_control` command.

When you specify an always-on attribute on a design object, to identify the always-on paths, the tool traverses back until it encounters a top-level port or a cell that is neither an inverter nor a buffer. The tool then traverses forward until it encounters a power domain boundary, which is considered to be the endpoint of the always-on path.

Using Basic Gates as Isolation Cells

When your target library does not contain a complete set of isolation cells, you can use the basic two-input AND, OR, NAND, and NOR gates as isolation cells. This flexibility allows you to use these basic cells for their usual logic as well as for isolation logic. Only the following types of basic gates can be used as isolation cells:

- Two-input AND, OR, NAND, and NOR gates

- Two-input AND, OR, NAND, and NOR gates with one of the inputs inverted

To enable this feature, you must set the `mv_use_std_cell_for_isolation` variable to `true`. You must then set the following attributes using the `set_attribute` command.

- Set the library cell-level attribute `ok_for_isolation` to `true` on the library cell.

This attribute denotes that the library cell can be used as a standard logic cell as well as an isolation cell. The following example shows how to set the `ok_for_isolation` attribute on the library cell A:

```
set_attribute [get_lib_cells lib_name/A] ok_for_isolation true
```

- Set the `isolation_cell_enable_pin` attribute to `true` on the library cell pin. This attribute specifies the pin to be used as the control pin of the isolation cell.

The following example script shows how to set the `isolation_cell_enable_pin` attribute to `true` on the in pin of the library cell A:

```
dc_shell> set_attribute [get_lib_pins lib_name/A/in] \
isolation_cell_enable_pin true
```

Additional Commands to Support Multivoltage Designs

This section describes commands that are not UPF standard supported commands, but commands supported by Power Compiler to support multivoltage design implementation and checking.

create_voltage_area

The `create_voltage_area` command creates a voltage area at the specified region for providing placement constraints of cells associated with the region. This command also enables you to create a voltage area of a specific geometry. The syntax of the `create_voltage_area` command is as follows:

```
create_voltage_area [-name name]
  [-coordinate {llx1 lly1 urx1 ury1 llx2 lly2 urx2 ury2 ...}]
  [-power_domain power_domain_name]
  [-guard_band_x integer_value]
  [-guard_band_y integer_value]
  [-color value]
  [-cycle_color]
  [-is_fixed]
  [-target_utilization float_value]
  [modules]
```

You can use the `-coordinate` option to define a target rectangular placement area for the voltage area. The geometry of the voltage area can be a rectangle or a rectilinear polygon.

Note:

The `create_voltage_area` command is supported only in Design Compiler topographical. Design Compiler topographical mode does not use any of the options other than `-coordinate`.

hookup_retention_register

This command hooks up the save and restore pins of the retention register to the save and restore signals respectively of the retention register. The save and restore signals are the signals specified using the `set_retention_control` command. This command works on the entire design. Power Compiler finds all the retention registers in the design and hooks up the restore and save pins to the appropriate control signals.

You use this command only in the bottom-up approach when your retention registers are already connected to the ports of your hierarchical block and you need to extend this connection to the top-level ports while synthesizing your top-level design.

Reporting Commands in the UPF Flow

The following reporting commands and checks are supported in Power Compiler. These are not UPF standard specified commands.

report_power_domain

The `report_power_domain` command reports the details of the specified power domain. The syntax of the `report_power_domain` command is as follows:

```
report_power_domain [domain_name]
```

When the power domain name is not specified, all power domains in the design are reported.

report_level_shifter

The `report_level_shifter` command reports the details of the level-shifter cells in the specified power domain. The details include the level-shifter cell names, the input and output power net information, violating level-shifter cells, and so on. With the `-verbose` option, this command reports the level-shifter strategy

report_power_switch

The `report_power_switch` command reports all the power switches in the specified power domain. The syntax of the `report_power_switch` command is as follows:

```
report_power_switch -domain domain_name
```

Use the `-domain` option to specify the power domain for which power switches are to be reported. If the specified power domain does not exist in the current scope, the `report_power_switch` command fails.

report_pst

The `report_pst` command reports the power state tables in the current design. The syntax of the `report_pst` command is as follows:

```
report_pst
  [-width line_width]
  [-significant_digits significant_digits]
  [-column_space column_space]
  [-tace_name]
  [-verbose]
  [-compress]
  [-power_nets supply_nets]
  [supplies supply_list]
  [-scope instance_name]
  [-derived]
```

The `report_pst` command reports the current power state tables. The report contains all the legal states of the power state table. The `-power_nets` option lists the supply nets to be included in the report. The order in which power nets are reported is determined by the order in which the nets are specified to the `-power_nets` option. When this option is not used all supply nets in the current design are reported. When you use the `-compress` option one entry of the report contains several power states combined together using wildcard character. For more information, see the `report_pst` command man page.

report_isolation_cell

The `report_isolation_cell` command reports all the isolation cells in the current scope. With the `-strategy` option, this command reports the details of the isolation strategy.

report_retention_cell

The `report_retention_cell` command reports the retention cells in the design. With the `-verbose` option, this command reports the list of the retention cells, the save and restore signals, and the retention strategy.

report_supply_net

The `report_supply_net` command reports the detailed information about the specified supply nets. The `-include_exception` option reports exceptional pins on the power net, if any.

report_supply_port

The `report_supply_port` command reports details of all the specified supply ports or all the supply ports in the current scope. The details of the supply port includes its full name, the scope it is created in, its direction, its supply state, and the supply net to which it is connected. Supply ports that are present on the power switches can also be reported.

report_target_library_subset

Use this command to find out the target library constraints, that is, to determine or confirm which target library subsets are assigned to which design instances.

check_mv_design

Use this command to check for design errors, including multivoltage constraint violations, electrical isolation violations, connection rules violations, and operating condition mismatches. Two switches let you control the level of information detail and limit the number of messages printed to the log file. Other switches, such as `-power_nets`, `-isolation`, `-level_shifters`, `-connection_rules`, `-opcond_mismatches`, and `-target_library_subset`, let you select among the available checking reports.

If you do not specify any checkers, violations are reported from all supported checkers. If you use the `-verbose` switch, the checkers provide detailed reports. If you omit this switch, summary reports are provided. This command includes the following arguments:

Argument	Description
<code>-verbose</code>	Optional. Provides a detailed report. If you do not use this option, a summary of any violations is reported.
<code>-max_messages</code> <i>message count</i>	Optional. Sets a limit, given by <i>message count</i> , on the number of messages per checker printed in the log file. If no checkers are specified, this is the message limit for all checkers. If you do not use this option, all messages are printed.
<code>-isolation</code>	Optional. Provides a report on electrical isolation errors with respect to power domains.
<code>-level_shifters</code>	Optional. Provides a report on all existing level shifters and connecting nets. Checks against the specified level-shifter strategy and threshold.
<code>-connection_rules</code>	Optional. Reports violations in always-on synthesis and pass-gate connections.
<code>-opcond_mismatches</code>	Optional. Reports incompatible operating conditions between instantiated technology cells and the cells' parent design.
<code>-target_library_subset</code>	Optional. Reports inconsistent settings among target libraries, target library subsets, and operating conditions.
<code>-power_nets</code>	Optional. Reports summary of power and ground connection; power and ground connections that cannot be derived and the reason for the same. Power and ground connections that do not match with the derived power and ground connection of the power domain.

For more details refer to the command man page.

Hierarchical UPF Design Methodology

Design Compiler topographical mode supports hierarchical UPF design methodology for multivoltage designs. This section describes the UPF portion of the hierarchical design methodology. For basic information on hierarchical design methodology, see the *Design Compiler User Guide*.

The hierarchical UPF methodology supported in Design Compiler is compatible with the hierarchical UPF methodology supported in IC Compiler. You must follow the guidelines to implement your designs using the hierarchical UPF methodology.

Guidelines

Follow this methodology and the interface guidelines to implement your multivoltage designs successfully using the hierarchical UPF flow.

Methodology Guidelines

You should follow these guidelines to synthesize your design using the UPF-based hierarchical flow.

1. Align the physical partitions with the power domain boundaries, as defined in the logical netlist.
2. Define the UPF constraints such that each power domain is at the same level of hierarchy as the element it contains.
3. Create all the supply nets for a power domain such that they are inside the power domain and are connected from the top level.
4. Specify the voltage for each supply net.
5. Specify the timing constraints as recommended in the *Design Compiler Hierarchical Reference Methodology* SolvNet article 026172.

Interface Guidelines

In the hierarchical implementation of a design, you first determine the physical partition. Follow these guidelines while partitioning your design.

- The partition boundary must correspond to a power domain boundary. The scope of the power domain must be at the same level of hierarchy as the partition or the module.
- The scopes of all power domains within a partition must be contained inside the partition.
- For all supply connections inside a partition, supply nets must be specified within the partition.
- The partitions should not be nested.

Understanding the Hierarchical UPF Design Methodology

To implement your design using the Design Compiler hierarchical UPF design methodology, follow these two steps.

1. [Block-Level Implementation](#)
2. [Top-Level Implementation](#)

Each of these steps is described in detail in the following sections.

Block-Level Implementation

Creating the Blocks

You create the block-level and top-level UPF files for your design. To create the blocks, you can use either the top-down approach or the bottom-up approach. The bottom-up approach is recommended because this determines the smallest block that can be compiled independently.

When the individual blocks and the top are synthesized, you can assemble the design either in Design Compiler or in IC Compiler. To assemble the design using IC Compiler, the tool requires the complete design database for the design planning stage. For more details, see [“Assembling Your Design” on page 12-37](#).

Generating the Block-Level UPF Constraints

To implement your hierarchical design using the hierarchical UPF methodology, your constraint specification in the UPF file must also be hierarchical. You can choose one of the following two ways to create the block-level and top-level UPF files. However, to use either of these methods, the power domains and the constraints must be properly aligned with the physical partitions.

- Write the power intent manually in the UPF file for all the blocks, including the top. If required, write the boundary constraints for the blocks.
- Use the `characterize` command to create the block-level power UPF constraints as well as the boundary constraints from the full chip UPF description. It is important to remember the following points when you use the `characterize` command to generate the block-level UPF constraints:
 - If your design does not have the control signals at the block-level interfaces and you cannot modify your block level interfaces, you must use the `characterize` command to generate the block-level UPF constraints.
 - By default, the `characterize` command translates the UPF constraints in the top design to the subblock.

However, if you use this approach, you will be able to perform equivalence checking only on the entire design, and not on each hierarchical block.

Note:

In this section, all necessary power management control signals are created manually in the hierarchical design. They are manually brought into the appropriate block-level interfaces. This is the recommended approach.

Using Manually Created Block-Level UPF Files

Your design should conform to the bottom-up methodology. Each block and its power intent in the UPF file must be written such that each block can be simulated and synthesized independently. You might have to write the boundary constraints for the blocks to capture any port that does not operate at the same voltage as the rest of the block. If a block contains a power domain, the UPF constraints refer to objects and power supplies only within the block.

Using Design Compiler Generated Block-Level UPF files

If you use the top-down approach to write your design or if your UPF file is nonmodular, Design Compiler can generate the block-level UPF using the `characterize` command. For the tool to correctly generate the block-level UPF file, your power domain definition and partitioning should comply with the guidelines mentioned in [“Interface Guidelines” on page 12-33](#). The UPF objects in the block should not refer to any object that is above the block in the hierarchy. You should follow these steps to synthesize your design using the hierarchical UPF design methodology:

1. Read the design and the UPF constraints for the entire design.
2. Specify the operating voltages for the supply nets and specify the timing constraints.
3. For each subblock in the design, perform the following tasks:
 - a. Execute the `characterize` command. Specify the block to be characterized as the argument to the `characterize` command. This command pushes the appropriate timing and power constraints from the top-level to the specified block,

When you characterize a block using the `characterize` command, the block-level power constraints as well as the boundary constraints that are specified by the `set_related_supply_net` command are set on the specified block.
 - b. Save the characterized block and the design data. Set the characterized block as the current instance and use the `write` command to save the characterized block. The command sequence is shown in the following example.

```
characterize BlockA
set current_instance BlockA
write -f ddc -hierarchy -output BlockA.characterized.ddc
```

- c. Remove the block from the top level using the `remove_design -hierarchical` command. When you remove the block, the UPF constraints associated with the block are also removed.
4. When all the subblocks have been characterized, saved in the .ddc file format, and removed, save the top-level design in the .ddc file format.

Synthesizing the Blocks

To synthesize each subblock of the hierarchical design, you can read the design in one of the following two methods:

- The RTL file and the manually written UPF file for each block
- GTECH netlist in the .ddc file for each block, written after the characterization step.

The difference between the two is the readability of the block-level UPF and the automatic inclusion of boundary constraints when you use the .ddc file generated after the characterization step and the ability to perform hierarchical verification using Formality. The power intent created by the `characterize` command is the same as the manually created UPF file. If you use the RTL design and the manually written UPF file, you should create appropriate boundary constraints.

You then use either the top-down or bottom-up synthesis flow options supported in Design Compiler topographical mode to perform block-level synthesis. For more details, see the SolvNet article 021034, *Hierarchical Flow Support in Design Compiler Topographical Mode*.

Top-Level Implementation

While performing top-level synthesis, you can read the top-level design in one of the following ways:

- The RTL and the UPF files for the top-level design
- The GTECH netlist in the .ddc file format, obtained after removing all the characterized subblocks.

Follow these steps to perform the top-level synthesis:

1. Read the block-level designs.
2. Read the top design.
3. For the synthesis step to obtain all the block-level constraints at the top-level, use the `propagate_constraints -power_supply_data` command.

If you are reading the design in ASCII format, you must read the UPF file using the `load_upf` command.

4. Synthesize the top-level design.

5. Save the synthesized design and the UPF constraints. When you save the design in the .ddc file, the UPF constraints are also saved in the .ddc file. You can save separately the UPF constraints in the ASCII format that you can use for equivalence checking.

Completing these steps completes the synthesis of your design using the Design Compiler hierarchical UPF flow. Using the synthesized design you can continue the flow in IC Compiler. For more details on assembling your design for the subsequent steps in IC Compiler see [“Assembling Your Design” on page 12-37](#).

Assembling Your Design

To continue with the hierarchical flow in IC Compiler, you can assemble your design either in Design Compiler or in IC Compiler. Note that you must explicitly ensure that the block-level UPF constraints are available in the top-level design during the optimization step of the top-level. You do this using the `propagate_constraints -power_supply_data` command. Use the following steps to assemble your design in Design Compiler for use in the further flow in IC Compiler:

1. Read all the synthesized subblocks.
2. Set the top-level design as the current design.
3. Link the design using the `link` command.
4. Use the `propagate_constraints -power_supply_data` command for all the block-level UPF constraints to be available at the top-level.
5. Save the design. This saved design is the full chip design database that you can use to start the design planning step in IC Compiler.

For more details, see the SolvNet article 026172, *IEEE 1801 (UPF) based Design Compiler Topographical Technology and IC Compiler Hierarchical Design Methodology*.

13

Power Optimization in the Multivoltage Domain

As described in [Chapter 11, “Multivoltage Design Concepts,”](#) each design block in a multivoltage design operates at its own operating voltage and is assigned to a voltage domain. You can define each block’s voltage level by specifying the operating conditions for the block.

Every voltage must correspond to one or more wholly contained logical hierarchies. All cells in a given hierarchy operate at the same voltage except for level shifter cells at the hierarchy interface.

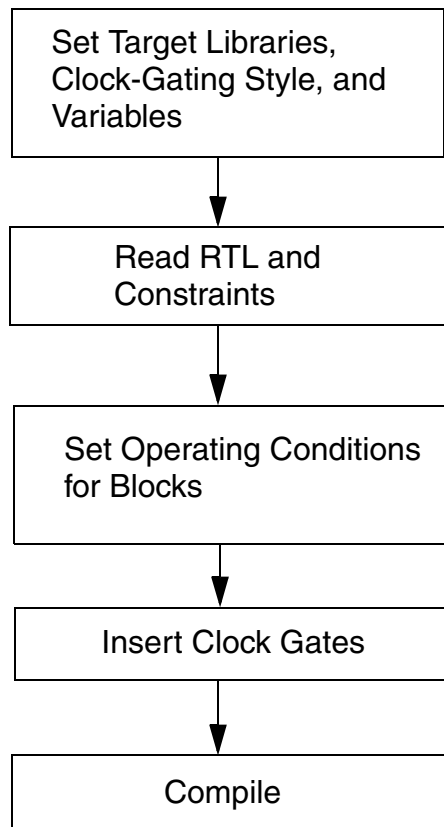
Power optimization flows within the multivoltage domain are described in the following sections:

- [Clock-Gating Flow](#)
- [Power-Gating Flow](#)
- [Reporting](#)

Clock-Gating Flow

[Figure 13-1](#) shows the clock-gating flow for multivoltage designs:

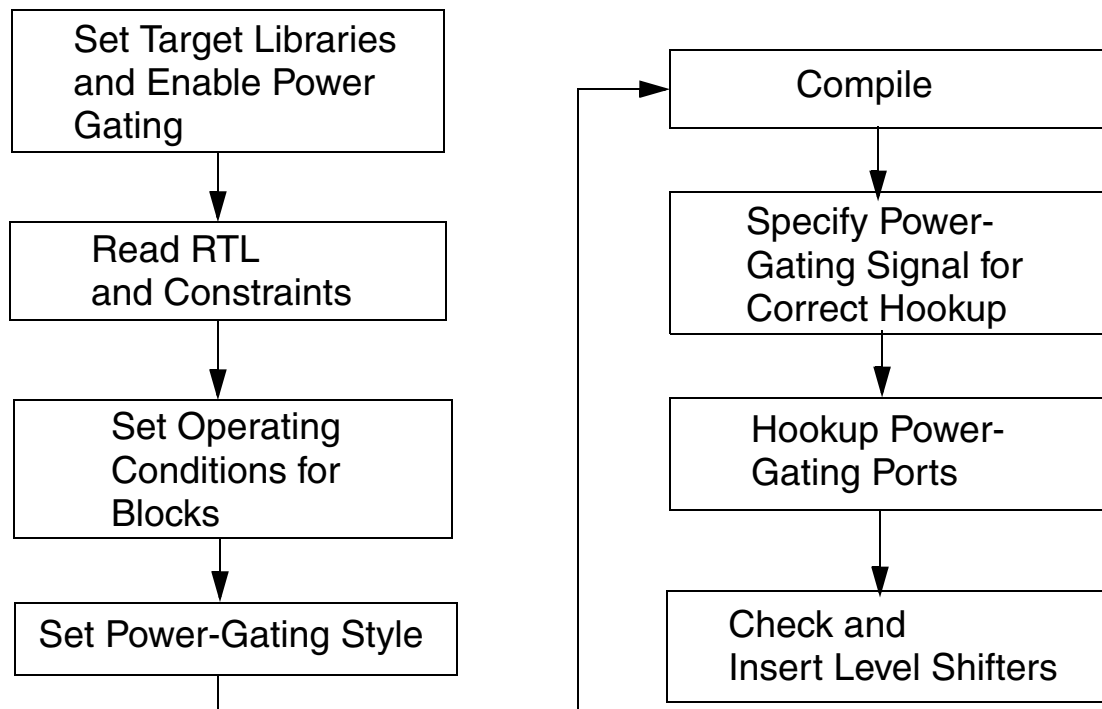
Figure 13-1 Clock-Gating Flow



If clock gating includes test logic, then use the `insert_clock_gating` command to hookup all the integrated clock-gate test pins to the top-level. This command automatically inserts level shifters on multivoltage designs. For information, see [“Inserting Buffer-Type Level-Shifter Cells”](#) in [Appendix C](#).

Power-Gating Flow

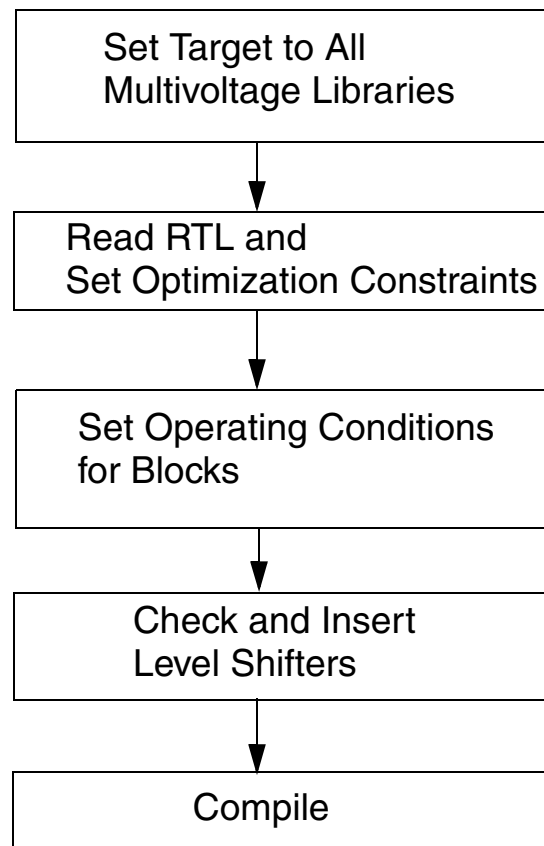
In Power Compiler, the power-gating feature is supported for multivoltage designs as shown in [Figure 13-2](#).

Figure 13-2 Power-Gating Flow

In this flow, the `insert_level_shifters` command is required after `hookup_power_gating_ports`. For information, see [“Inserting Buffer-Type Level-Shifter Cells” in Appendix C](#).

Gate-Level Power Optimization Flow

All dynamic and leakage optimizations are supported by Power Compiler for multivoltage designs. [Figure 13-3](#) shows a one-pass leakage optimization flow.

Figure 13-3 Gate-Level Power Optimization Flow

Reporting

The following sample report shows the output for the `report_clock_gating -gating` command, which reports operating conditions and voltage of the gates.

```

-----
                        Clock Gating Cell Report
-----

Clock Gating Bank : clk_gate_out1_reg (ss_hvt_0v70_125c:
0.7)
-----

STYLE = latch, MIN = 2, MAX = unlimited, OBS_DEPTH = 5

INPUTS :
clk_gate_out1_reg/CLK = clk

```

```

clk_gate_out1_reg/EN = N6
clk_gate_out1_reg/TE = test_se

OUTPUTS :
clk_gate_out1_reg/ENCLK = net107

Clock Gating Bank : sub/clk_gate_out_reg (ss_hvt_1v08_125c:
1.08)
-----

STYLE = latch, MIN = 2, MAX = unlimited, OBS_DEPTH = 5

INPUTS :
sub/clk_gate_out_reg/CLK = n22
sub/clk_gate_out_reg/EN = N6
sub/clk_gate_out_reg/TE = test_se

OUTPUTS :
sub/clk_gate_out_reg/ENCLK = net95

```

The following example shows the output for the `report_clock_gating -gating -ver` command.

```

Clock Gating Bank : clk_gate_out1_reg (ss_hvt_0v70_125c:
0.7)
-----

STYLE = latch, MIN = 2, MAX = unlimited, OBS_DEPTH = 5

TEST INFORMATION :
OBS_POINT = NO, CTRL_SIGNAL = scan_enable, CTRL_POINT =
before

INPUTS :
clk_gate_out1_reg/CLK = clk
clk_gate_out1_reg/EN = N6
clk_gate_out1_reg/TE = test_se

OUTPUTS :
clk_gate_out1_reg/ENCLK = net107

RELATED REGISTERS :
out1_reg[3] (ss_hvt_0v70_125c: 0.7)
out1_reg[2] (ss_hvt_0v70_125c: 0.7)
out1_reg[1] (ss_hvt_0v70_125c: 0.7)
out1_reg[0] (ss_hvt_0v70_125c: 0.7)

```

The following example shows the output for the `report_clock_gating -gated` command.

```

-----
Gated Register Report

```

Clock Gating Bank Voltage	Gated Register	Opond	
clk_gate_out1_reg		ss_hvt_0v70_125c	0.7
	out1_reg[3]	ss_hvt_0v70_125c	0.7
	out1_reg[2]	ss_hvt_0v70_125c	0.7
	out1_reg[1]	ss_hvt_0v70_125c	0.7
	out1_reg[0]	ss_hvt_0v70_125c	0.7
sub/clk_gate_out_reg		ss_hvt_1v08_125c	
1.08			
	sub/out1_reg[3]	ss_hvt_1v08_125c	1.08
	sub/out1_reg[2]	ss_hvt_1v08_125c	1.08
	sub/out1_reg[1]	ss_hvt_1v08_125c	1.08
	sub/out1_reg[0]	ss_hvt_1v08_125c	1.08
	sub/r1/out_reg[3]	ss_hvt_1v08_125c	1.08
	sub/r1/out_reg[2]	ss_hvt_1v08_125c	1.08
	sub/r1/out_reg[1]	ss_hvt_1v08_125c	1.08
	sub/r1/out_reg[0]	ss_hvt_1v08_125c	1.08
sub/r0/clk_gate_out_reg		ss_hvt_0v90_125c	
0.9			
	sub/r0/out_reg[3]	ss_hvt_0v90_125c	0.9
	sub/r0/out_reg[2]	ss_hvt_0v90_125c	0.9
	sub/r0/out_reg[1]	ss_hvt_0v90_125c	0.9
	sub/r0/out_reg[0]	ss_hvt_0v90_125c	0.9

The following example shows `report_power -cell`, which includes information about the operating voltage, condition, and voltage.

```
*****
Report : power
-cell
-analysis_effort low
-sort_mode cell_internal_power
Design : top
Version: Y-2006.06-H03
Date : Mon Apr 10 16:30:31 2006
*****

Library(s) Used:

slow_0v70_1v08 (File: /remote/testcases13/
9000109001_9000109500/9000109260/
LIB/slow_0v70_1v08.db)
ss_hvt_0v70_125c (File: /remote/testcases13/
9000109001_9000109500/9000109260/
LIB/ss_hvt_0v70_125c.db)
slow_0v90_1v08 (File: /remote/testcases13/
```

```

9000109001_9000109500/9000109260/
LIB/slow_0v90_1v08.db)
  ss_hvt_1v08_125c (File: /remote/testcases13/
9000109001_9000109500/9000109260/
LIB/ss_hvt_1v08_125c.db)
  slow_0v70_0v90 (File: /remote/testcases13/
9000109001_9000109500/9000109260/
LIB/slow_0v70_0v90.db)
  ss_hvt_0v90_125c (File: /remote/testcases13/
9000109001_9000109500/9000109260/
LIB/ss_hvt_0v90_125c.db)

```

Operating_conditions:

```

  ss_hvt_0v70_125c Library: ss_hvt_0v70_125c Voltage: 0.7
  ss_hvt_1v08_125c Library: ss_hvt_1v08_125c Voltage: 1.08
  ss_hvt_0v90_125c Library: ss_hvt_0v90_125c Voltage: 0.9

```

Wire Load Model Mode: top

```

Global Operating Voltage = 0.7
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 = 1pW

```

Attributes

h - Hierarchical cell

Cell Driven Net Tot Dynamic Cell

Internal Switching Power Leakage

Cell Power Power (% Cell/Tot) Power Attrs Library Opcond

Voltage

```

-----
sub 0.6076 N/A N/A (N/A) 602109.1875
  h ss_hvt_1v08_125c 1.08
  clk_gate_out1_reg 0.0357 N/A N/A (N/A) 4511.1348 h
  ss_hvt_0v70_125c 0.7
  U7 0.0175 2.391e-03 1.99e-02 (88%) 35692.0625 slow_0v70_1v08
  slow_0v70_1v08 0.7
  -&lt; 1.08(VDDH:1.08,VDDL:0.70,VSS:0.00)
  out1_reg[0] 6.249e-03 0.0000 6.25e-03 (100%) 1722.0234
  ss_hvt_0v70_125c
  ss_hvt_0v70_125c 0.7
  out1_reg[1] 6.245e-03 0.0000 6.25e-03 (100%) 1722.0342
  ss_hvt_0v70_125c

```

```
ss_hvt_0v70_125c 0.7
out1_reg[2] 6.243e-03 0.0000 6.24e-03 (100%) 1721.9194
ss_hvt_0v70_125c
ss_hvt_0v70_125c 0.7
out1_reg[3] 6.233e-03 0.0000 6.23e-03 (100%) 1721.6326
ss_hvt_0v70_125c
ss_hvt_0v70_125c 0.7
-----
Totals (7 cells) 685.826uW N/A N/A (N/A) 649.200nW
```


14

Multicorner-Multimode Optimization

This chapter describes the support for multicorner-multimode technology in Design Compiler Graphical, in the following sections:

- [Basic Multicorner-Multimode Concepts](#)
- [Basic Multicorner-Multimode Flow](#)
- [Setting Up the Design for a Multicorner-Multimode Flow](#)
- [Handling Libraries in the Multicorner-Multimode Flow](#)
- [Automatic Inference of Operating Conditions for Macro, Pad and Switch Cells](#)
- [Scenario Management Commands](#)
- [Power Optimization Techniques](#)
- [Reporting Commands](#)
- [Supported SDC Commands](#)
- [Multicorner-Multimode Script Example](#)

Basic Multicorner-Multimode Concepts

Present-day designs are often required to operate under multiple operating conditions or 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.

A corner is defined as a set of libraries characterized for process, voltage and temperature 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.

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 on.

Scenario Definition

A *scenario* is a mode or a corner or a combination of both that can be analyzed and optimized. Optimization of multicorner-multimode design involves managing the scenarios of the design. For more details on scenario management, see [“Scenario Management Commands” on page 14-15](#).

Multicorner-Multimode Optimization

The multicorner-multimode feature in Design Compiler Graphical provides compatibility between flows in Design Compiler and IC Compiler.

Supported Features

The following are the important points about the support of this features:

- Multicorner-multimode technology is supported only in topographical mode with the DC-Extension license.
- Dynamic and leakage power optimizations are supported
- Only the UPF flow is supported for the multivoltage designs
- All options of `compile_ultra` command are supported

Unsupported Features

The following features are not supported in Design Compiler Graphical for multicorner-multimode designs

- The scenario management command, `set_scenario_options`, is not supported. You can use the `set_max_leakage_power` command to perform leakage power optimization in specific scenarios.
- 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 in the power reports. So power reports such as those generated by the `report_power` command do not include details of estimated clock tree power.
- k-factor scaling. (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.)
- Using the `set_min_library` command in a scenario-specific way is not supported. (The `set_min_library` command applies to all scenarios. Therefore, if you use the command to relate a minimum library to a specific maximum library, the relationship applies to all scenarios.)

Concurrent Multicorner-Multimode Optimization and Timing Analysis

Concurrent 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 DRC. As a result, the timing and constraint reports show worst-case timing across all scenarios.

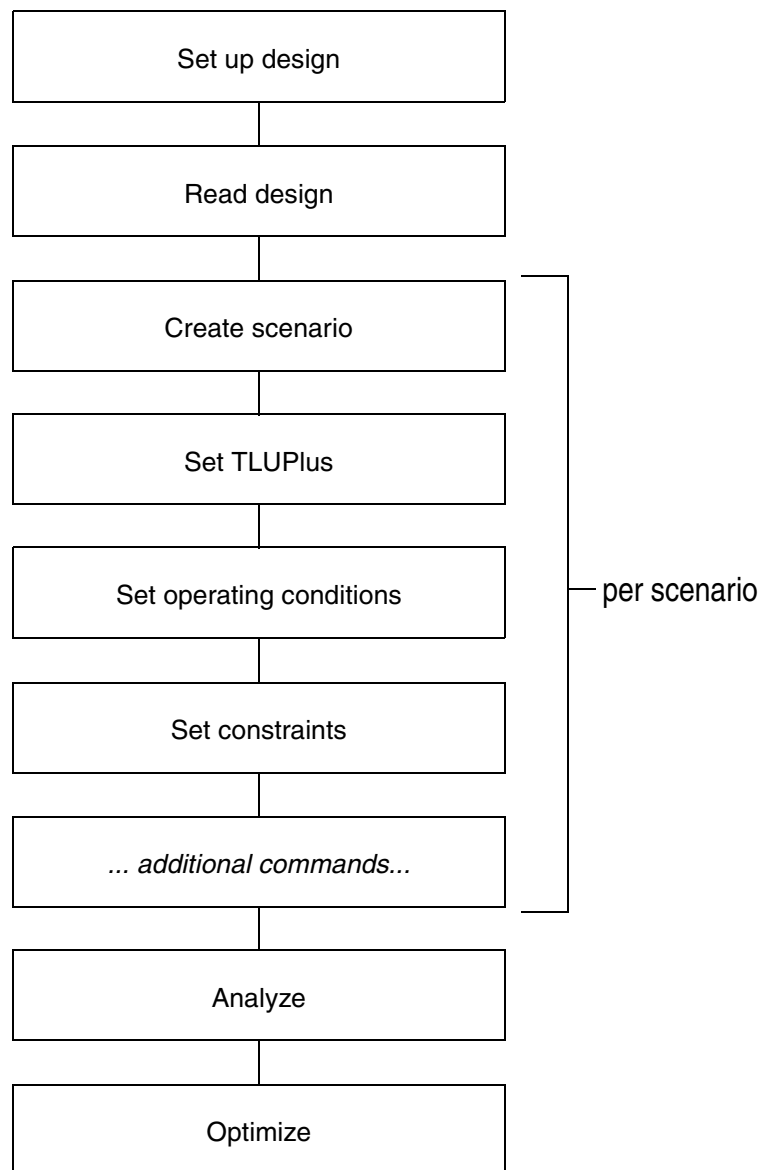
Timing analysis can be performed using, one of the following two methods:

- Traditional minimum or maximum methodology
- Early-late analysis, such as that in PrimeTime, utilizing the on-chip variation (OCV) switch in the `set_operating_conditions` command.

Basic Multicorner-Multimode Flow

Figure 14-1 shows the basic multimode flow.

Figure 14-1 Basic Multicorner-Multimode Flow



Multicorner-multimode optimization involves managing the scenarios. You use the `create_scenario` command to create the scenarios. You can create multiple scenarios, and for each scenario, you can set constraints specific to the mode and operating conditions specific to the corner. After you configure the scenarios, you can optionally activate a subset of these scenarios, using the `set_active_scenarios` command.

As shown in [Figure 14-1](#), a scenario definition usually includes 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 the `read_sdf` command to set the correct net RC and pin-to-pin delay information in the respective scenarios.

Setting Up the Design for a Multicorner-Multimode Flow

To set up a design for a multicorner-multimode flow, you must specify the TLUPlus files, operating conditions, and Synopsys Design Constraints as shown in [Figure 14-1 on page 14-4](#). 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 file settings explicitly for each scenario. If a TLUPlus setup is not correct, the tool issues the following error message:

```
Error: TLU+ sanity check failed (OPT-1429)
```

To allow for temperature scaling, the TLUPlus files must contain the `GLOBAL_TEMPERATURE`, `CRT1`, and (optionally) `CRT2` variables. 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
...

```

The TLUPlus file settings, which you specify by using the `set_tlu_plus_files` command, must be made explicitly for each scenario. If a TLUPlus setup is not correct, an error similar to the following message is issued:

```
Error: tlu_plus files are not set in this scenario s1.  
      RC values will be 0.
```

Specifying Operating Conditions

The operating condition of the design must be set for each scenario. You can specify different operating conditions for different scenarios using the `set_operating_condition` command.

```
dc_shell-topo> set_operating_conditions SLOW_95 -library  
  
max_vmax_v95_t125
```

If an operating condition is not defined for a particular scenario, MV-020 and MV-21 warnings are issued.

Specifying Constraints

In a multicorner-multimode design, you are required to specify Synopsys Design Constraints (SDC) specific to a scenario after you have created the scenario. Any scenario-specific constraints that existed before are discarded, as shown in the following example:

```
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 properly in multicorner-multimode designs:

- [Link Libraries With Equal Nominal PVT Values](#)
- [Unsupported k-factors](#)
- [Automatic Detection of Driving Cell Library](#)
- [Relating the Minimum Library to the Maximum Library](#)

- [Unique Identification of Libraries Based on File Names](#)

Link Libraries With Equal Nominal PVT Values

The link library lists all of 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.

Library grouping is based on their PVT values. 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. If you do not correct this error, no optimization is performed.

Link Library Example

[Table 14-1](#) shows the libraries in the link library list, their nominal PVT values, and the operating condition (if any) specified in each library. The design has instances of combinational, sequential, and macro cells.

Table 14-1 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 a scenario s1 with the 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.db:slow
```

Note that providing the `-library` option in the `set_operating_conditions` command merely 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 linking scheme, you can link libraries that do not have operating condition definitions. The scheme also provides the flexibility of having multiple library files (for example, one 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 (same name and order), a warning is issued, stating that the libraries are inconsistent.

You should run the `check_library` command before running a multicorn-multimode flow, as shown in the following example:

```
set_check_library_options -mcm
check_library -logic_library_name {a.db b.db}
```

When you use the `-mcm` option with the `set_check_library_options` command, the `check_library` command performs multicorn-multimode specific checks such as the operating condition or power-down consistencies. When inconsistencies are detected, the tool generates a report. In addition, the tool also issues the following summary information message:

```
Information: Logic library consistency check FAILED for MCM.
(LIBCHK-360)
```

To overcome the LIBCHK-360 messages, you must check the libraries and the report to identify the cause for the inconsistency. The man page of the LIBCHK-360 information message describes possible causes for the various 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.

Distinct PVT Requirements

If the maximum libraries associated with each corner (scenario) do not have distinct PVT values, the cell instances are linked incorrectly, which results 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), 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 values but not the library to link.

The following two tables and the following script demonstrate the problem:

[Table 14-2](#) shows the libraries in the link library, listed *in order*, their nominal PVT values; and the operating condition that is specified in each library.

Table 14-2 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 14-3](#) and the script commands that follow show the operating condition specification for each of the scenarios.

Table 14-3 Scenarios and Their Operating Conditions

	Scenarios			
	s1	s2	s3	s4
Maximum Operating Condition (Library)	WORST (Typ.db)	WORST (TypHV.db)	WORST (Ftyp.db)	WORST (Typ.db)
Minimum Operating Condition (Library)	None	None	None	BEST (HoldTyp.db)

```
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
```

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 linked to the library cells in the Ftyp.db library, assuming that all the libraries include the library cells required to link the design.

Ambiguous Libraries Warning

When you use multiple libraries, if any of the libraries with same-name cells have the same nominal PVT, a warning is issued, stating that the libraries are ambiguous. The warning also states which libraries are being used and which are being ignored.

Unsupported k-factors

Multicorner-multimode design libraries do not support the use of 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.

Relating the Minimum Library to the Maximum Library

The `set_min_library` command is not scenario-specific. This implies that if you use this command to relate a minimum library to a particular maximum library, that relationship applies to all scenarios.

Table 14-4 Unsupported Multiple Minimum Library Configuration

	Scenarios	
	s1	s2
Maximum library	Slow.db	Slow.db
Minimum library	Fast_0yr.db	Fast_10yr.db

For example, you could not relate two different minimum libraries—for example, `Fast_0yr.db` and `Fast_10yr.db` – with the maximum library, `Slow.db`, in two separate scenarios. The first minimum library that you specify would apply to both scenarios. [Table 14-4](#) shows the *unsupported* configuration.

Note, however, that a minimum library can be associated with multiple maximum libraries. As shown in the example in [Table 14-5](#), 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 14-5 Supported Minimum-Maximum Library Configuration

	Scenarios	
	s1	s2
Maximum Library	Slow.db	SlowHV.db
Minimum Library	Fast_0yr.db	Fast_0yr.db

Unique Identification of Libraries Based on File Names

Two libraries with the same name can be uniquely identified if their file names, which precede the library names, which are colon-separated, are unique. For example, the library ABC.db:stdcell (where ABC.db is the library file name and stdcell is the library name) is identifiable with respect to the library DEF.db:stdcell.

However, two libraries that have the same file name and library name but reside in different directories are not uniquely distinguishable. The following two libraries are not uniquely distinguishable:

```
/remote/snps/testcase/LIB/fast/ABC.db
```

```
/remote/snps/testcase/LIB/slow/ABC.db
```

Automatic Inference of Operating Conditions for Macro, Pad and Switch Cells

In multivoltage and multicorner-multimode designs, as designs increase in size and complexity, manually specifying the operating conditions and linking them with the appropriate library cells with matching operating conditions becomes tedious and time consuming. So in such designs it is useful to automatically infer the operating conditions, especially for the multi-rail pad cells, macro cells and switch cells. When the operating condition set on the design does not match the operating condition of the cell rails or when the design operating condition does not have rails at all, Power Compiler issues a LIBSETUP-001 error message.

Power Compiler can infer the operating conditions for macro cells, pad cells, and switch cells in both UPF and non-UPF modes. However, you have to set the following variables appropriately for the tool to infer the operating conditions.

```
libsetup_pad_opcond_inference_level
libsetup_macro_opcond_inference_level
libsetup_switch_opcond_inference_level
```

Note:

Power Compiler does not perform automatic operating condition inference for standard cells. The operating conditions of the standard cells should match exactly with the operating conditions of the design.

The value of these variables determine the degree to which the inferred operating condition can deviate from the operating condition of the design. When you permit higher deviation, the probability of automatic inference of operating condition is higher, resulting in a lesser number of LIBSETUP-001 error messages. The values that you can specify with these variables determines the level of deviation that you permit to the tool. The following table summarizes the values that you can specify and its impact on the automatic inference:

Value Specified with the Variable	Degree of Deviation in the Inferred Operating Condition and its Impact
EXACT	Operating condition inferred is exact. This will result in no inference at all. Timing is exact
UNIQUE_RESOLVED	The library cell whose name matches exactly with the cell is inferred. You cannot choose a different library cell. Timing will be incorrect. You do not encounter LIBSETUP-001 error messages
CLOSEST_RESOLVED	This is the default value. If multiple library cells are available, library cell whose operating condition is closest to the design is chosen. Choosing this operating condition can cause inaccurate timings
CLOSEST_UNRESOLVED	Similar to CLOSEST_RESOLVED. The library cell chosen can be less closer than when you set the value to CLOSEST_RESOLVED.

Power Compiler automatically infers the operating condition for each instance of a macro, pad or switch cell that does not have an explicitly specified operating condition. The automatic inference is performed when the operating condition on a macro, pad or switch cell does not match the operating condition set on the design and the tool detects a potential LIBSETUP-001 error.

The details of the behavior of the tool when you set a specific value to these variables are described in this section:

- EXACT

When you set the value to EXACT, the automatic operating condition inference is not performed.

- UNIQUE_RESOLVED

The tool performs a name based search in the target libraries. If multiple library cells match with the cell name, the tool does not perform the inference. However, if the cell is present in a unique library file and no other library contains the cell, the operating condition is inferred. If the library cell has explicit power and ground connections, and if the rail voltage matches the explicit power connection rail voltages, the operating condition is inferred. Otherwise, operating condition is not inferred on the cell and a LIBSETUP-001 error message is issued.

- CLOSEST_RESOLVED

This is the default value used when you do not use the variables.

For each macro, pad or switch cell instance, the tool finds the set of library cells with the same name. If multiple library cells match with the instance name, these library cells are filtered to choose a single library cell. Also, if the matching cells are connected to power nets, cells whose rail voltages do not match the explicit power connection rails are eliminated from further filtering. Within this set, the tool groups the library cells, taking into account these conditions, in the order of priority in which they are listed:

1. The PVT values of the library cell match the PVT values of the design.
2. The process, temperature and voltage values from one of the rails match the PVT values of the design.
3. The voltage and temperature values of the library cell match the process and voltage values of the design.
4. The temperature and voltage value of the library cell matches the temperature and voltage value of the design.
5. The process and voltage value of the library cell matches the process and voltage value of the design.
6. The voltage value of the library cell matches the voltage value of the design.
7. The voltage value from one of the rails matches the voltage value of the design.
8. The process and temperature value of the library cell matches the process and temperature value of the design.

9. None of process, voltage, and temperature values of the library cell matches with the process, voltage, and temperature values of the design.

After the library cells are grouped, the tool inspects each group in the order mentioned above. The inference is terminated in the following situations:

- None of the groups contain exactly one cell.
- None of the groups contain any library cell.

When Power Compiler finds a group that contains exactly one cell the tool chooses the library cell and uses the PVT values of that cell as the operating condition of the associated macro cell, switch cell or the pad cell.

- **CLOSEST_UNRESOLVED**

When you set the value of the variables to `CLOSEST_UNRESOLVED`, the tool groups the library cells based on the matching names, as in `CLOSEST_RESOLVED`. The tool then picks the first library cell from the first non-empty group of library cells. It then set the operating condition of the operating condition of the library cell on the specific cell instance and links the cell instance to the library cell.

The automatic inference of operating conditions is supported in both IEEE 1801™ (UPF) and non-UPF modes. You can disable the automatic inference of operating conditions by explicitly setting the operating conditions. The tool issues a LIBSETUP-751 information message when operating conditions are successfully inferred on a cell instance.

Scenario Management Commands

You use the following commands to create and manage scenarios:

- `create_scenario`
- `current_scenario`
- `all_scenarios`
- `set_active_scenarios`
- `all_active_scenarios`
- `remove_scenario`
- `report_scenarios`

For more details on the use of these commands, see the Design Compiler Topographical Technology chapter in the *Design Compiler User Guide*.

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 power optimization on multicorner-multimode designs.

Optimizing for Leakage Power

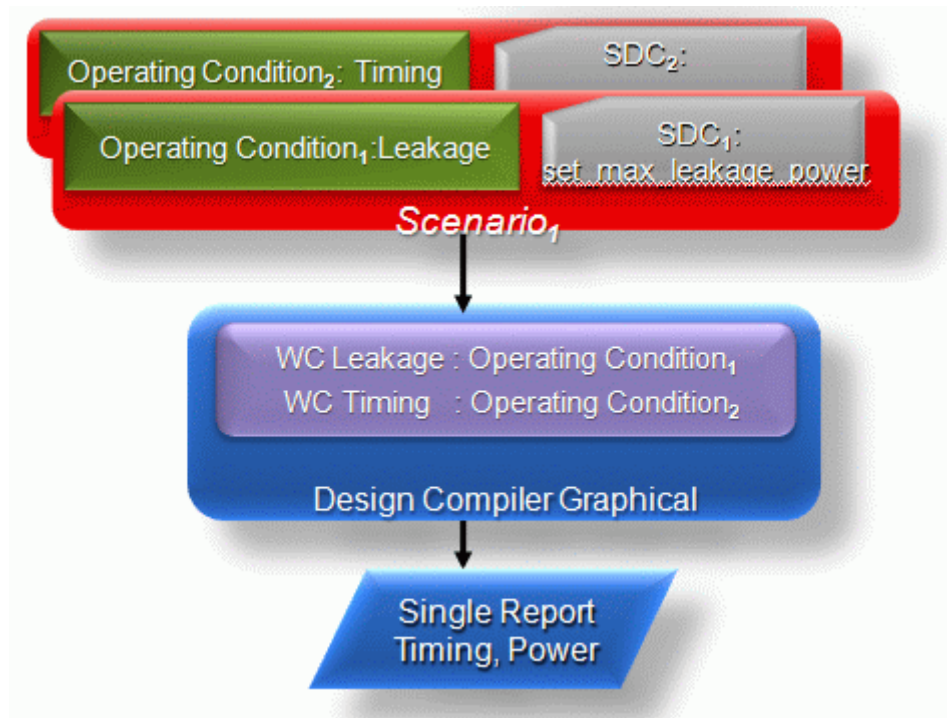
[Figure 14-2](#) shows how to set various constraints on different scenarios of a multicorner-multimode design.

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 14-2 Setting Different Constraints on Different Scenarios



Note the following points when you optimize for leakage power in multicorner-multimode designs:

- 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 14-1 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 to set dynamic power constraints on the scenarios of a multicorner-multimode design.

Example 14-2 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.

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

```

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 [all_scenarios]

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, the `report_constraint` command 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
*****

```

Group (max_delay/setup)	Cost	Weight	Weighted 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	

...

Constraint	Multi-Scenario 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 for each 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	Cell Internal Power (mW)	Driven Net Switching Power (mW)	Tot Dynamic Power (mW)	Cell Leakage (% 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 14-6 lists the SDC commands supported in the multicorn-multimode flow.

Table 14-6 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 14-3](#) show a basic sample script for the multicorner-multimode flow.

Example 14-3 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
```

The multicorner-multimode design in [Figure 14-3](#) and the subsequent examples scripts [Example 14-4](#) and [Example 14-5](#) show how you define your power intent in the UPF file and define the scenarios for a multicorner-multimode multivoltage design.

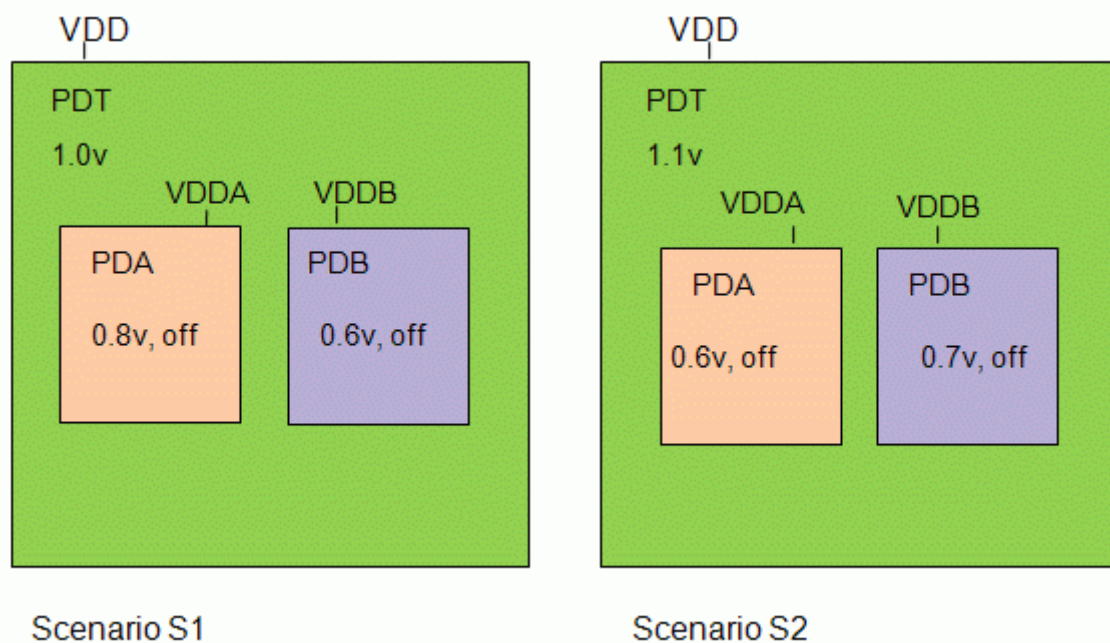
Multicorner-multimode multivoltage designs are useful in applications such as dynamic voltage and frequency scaling (DVFS). In hierarchical designs, the top-level design is generally optimized at a different voltage and in a different corner than the subdesigns of the hierarchy. The power intent specification can be for the entire design in a single UPF (Unified Power Format) file.

Standard cell and special cell libraries should be available to satisfy all voltages defined across multiple corners.

The design in [Figure 14-3](#) has two scenarios of operation, S1 and S2. In the scenario S1, the power domain PDT operates at 1.0V, while the power domain PDA operates at 0.8V or OFF and power domain PDB operates at 0.6V or OFF. In scenario S2, the power domain PDT operates at 1.1V, while the power domain PDA operates at 0.6V or OFF and power domain PDB operates at 0.7V or OFF.

Although the various subdesigns operate at different voltages, you need only a single UPF file to specify your power intent for the entire design and all its subdesigns. The specific voltages set on the supply nets are scenario-specific and are set by using the `set_voltage` command in each scenario.

Figure 14-3 Multicorner-Multimode Design with Multivoltage



[Example 14-4](#) and [Example 14-5](#) show sample scripts using the UPF flow for the multivoltage, multicorner-multimode design in [Figure 14-3](#).

Example 14-4 UPF File Describing Design Intent

```

Sample UPF File
## Create Power Domains
create_power_domain PDT -include_scope
create_power_domain PDA -elements PD_PDA
create_power_domain PDB -elements PD_PDB

## Create Supply Nets
create_supply_net VDD -domain PDT
create_supply_net VDDA -domain PDA
create_supply_net VDDB -domain PDB
create_supply_net VSS -domain PDT
create_supply_net VSS -domain PDA -reuse
create_supply_net VSS -domain PDB -reuse

## Create Supply Ports
create_supply_port VDD
create_supply_port VDDA
create_supply_port VDDB
create_supply_port VSS

## Connect supply nets
connect_supply_net VDD -ports VDD
connect_supply_net VDDA -ports VDDA
connect_supply_net VDDB -ports VDDB
connect_supply_net VSS -ports VSS

### Adding port states
add_port_state VDD -state {HV1 1} -state {HV2 1.1}
add_port_state VDDA -state {LV1 0.8} -state {LV3 0.6} -state {OFF off}
add_port_state VDDB -state {LV2 0.9} -state {LV4 0.7} -state {OFF off}
create_pst top_pst -supplies "VDD VDDA VDDB"
add_pst_state PM1 -pst top_pst -state { HV1 LV1 LV3 }
add_pst_state PM2 -pst top_pst -state { HV1 LV1 OFF }
add_pst_state PM3 -pst top_pst -state { HV1 OFF LV3 }
add_pst_state PM4 -pst top_pst -state { HV1 OFF OFF }
add_pst_state PM5 -pst top_pst -state { HV2 LV2 LV4 }
add_pst_state PM6 -pst top_pst -state { HV2 LV2 OFF }
add_pst_state PM7 -pst top_pst -state { HV2 OFF LV4 }
add_pst_state PM8 -pst top_pst -state { HV2 OFF OFF }

```

Example 14-5 Sample Tcl Script

```
load_upf example.upf    ## UPF file defined above

create_scenario s1
read_sdc s1.sdc
set_operating_conditions WCCOM lib1.0V
set_voltage -object_list VDD 1.0
set_voltage -object_list VDDA 0.8
set_voltage -object_list VDDB 0.9

create_scenario s2
read_sdc s2.sdc
set_operating_conditions BCCOM lib1.1V
set_voltage -object_list VDD 1.1
set_voltage -object_list VDDA 0.6
set_voltage -object_list VDDB 0.7
set_max_leakage_power 0
compile_ultra -scan -gate_clock
```

Note:

The UPF file is not scenario-specific. As a result, the UPF file must contain port state definitions and power state tables for all the scenarios.

You use the `load_upf` command to read the UPF script shown in [Example 14-4](#).

A

Integrated Clock-Gating Cell Example

This appendix contains an example .lib description of an integrated clock-gating cell and some schematic examples of rising (positive) and falling (negative) edge integrated clock-gating cells.

This appendix contains the following sections:

- [Library Description](#)
- [Sample Schematics](#)

Library Description

[Example A-1](#) is a description of an integrated clock-gating cell that demonstrates the following features:

- The `clock_gating_integrated_cell` attribute
- Appropriate clock-gating attributes on three pins
- Setup and hold arc on enable pin (EN) with respect to the clock pin (CP)
- Combinational arcs from enable pin (EN) and clock pin (CP) to the output pin (Z)
- State table and state function on the output pin (Z)
- Internal power table

Example A-1 HDL Description, Integrated Clock-Gating Cell

```
cell(CGLP) {
  area : 1;
  clock_gating_integrated_cell : "latch_posedge";
  dont_use : true;
  statetable(" CP EN ", "IQ ") {
    table : " L  L  : - : L ,\
              L  H  : - : H ,\
              H  -  : - : N ";
  }
  pin(IQ) {
    direction : internal;
    internal_node : "IQ";
  }
  pin(EN) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_enable_pin : true;
    timing() {
      timing_type : setup_rising;
      intrinsic_rise : 0.4;
      intrinsic_fall : 0.4;
      related_pin : "CP";
    }
    timing() {
      timing_type : hold_rising;
      intrinsic_rise : 0.4;
      intrinsic_fall : 0.4;
      related_pin : "CP";
    }
  }
  pin(CP) {
    direction : input;
```

```

    capacitance : 0.031419;
    clock_gate_clock_pin : true;
    min_pulse_width_low : 0.319;
}
pin(Z) {
    direction : output;
    state_function : "CP * IQ";
    max_capacitance : 0.500;
    max_fanout : 8
    clock_gate_out_pin : true;
    timing() {
        timing_sense : positive_unate;
        intrinsic_rise : 0.48;
        intrinsic_fall : 0.77;
        rise_resistance : 0.1443;
        fall_resistance : 0.0523;
        rise_resistance : 0.1443;
        fall_resistance : 0.0523;
        slope_rise : 0.0;
        slope_fall : 0.0;
        related_pin : "CP";
    }
    timing() {
        timing_sense : positive_unate;
        intrinsic_rise : 0.22;
        intrinsic_fall : 0.42;
        rise_resistance : 0.1443;
        fall_resistance : 0.0523;
        slope_rise : 0.0;
        slope_fall : 0.0;
        related_pin : "EN";
    }
    internal_power () {
        rise_power(li4X3){
            index_1("0.0150, 0.0400, 0.1050, 0.3550");
            index_2("0.050, 0.451, 1.501");
            values("0.141, 0.148, 0.256",\
                "0.162, 0.145, 0.234",\
                "0.192, 0.200, 0.284",\
                "0.199, 0.219, 0.297");
        }
        fall_power(li4X3){
            index_1("0.0150, 0.0400, 0.1050, 0.3550");
            index_2("0.050, 0.451, 1.500");
            values("0.117, 0.144, 0.246",\
                "0.133, 0.151, 0.238",\
                "0.151, 0.186, 0.279",\
                "0.160, 0.190, 0.217");
        }
        related_pin : "CP EN" ;
    }
}
}

```

When creating your model, examine whether it includes all the `clock_gate` attributes on both the cell and on the pins. The `check_dft` command and a few Power Compiler commands require these attributes in order to recognize the functionality of the cell. DFT Compiler does not recognize this cell. If these attributes are not included, an error message displays. Include the following attributes in your model:

- `clock_gating_integrated_cell`
- `clock_gate_test_pin`
- `clock_gate_enable_pin`
- `clock_gate_out_pin`
- `clock_gate_clock_pin`

Library Compiler can interpret the functionality of the integrated clock-gating cell directly from the state table and state function. The following example shows the `clock_gating_integrated_cell` attribute with a generic value.

```
cell(CGLP) {
  area : 1;
  clock_gating_integrated_cell : "generic";
  dont_use : true;
  statetable(" CP EN ", "IQ ") {
    table : " L L : - : L ,\
  L H : - : H ,\
  H - : - : N ";
  }
  pin(IQ) {
    direction : internal;
    internal_node : "IQ";
  }
  ...
  pin(Z) {
    direction : output;
    state_function : "CP * IQ";
    max_capacitance : 0.500;
    max_fanout : 8
    clock_gate_out_pin : true;
    timing() {
  ...
}
```

Sample Schematics

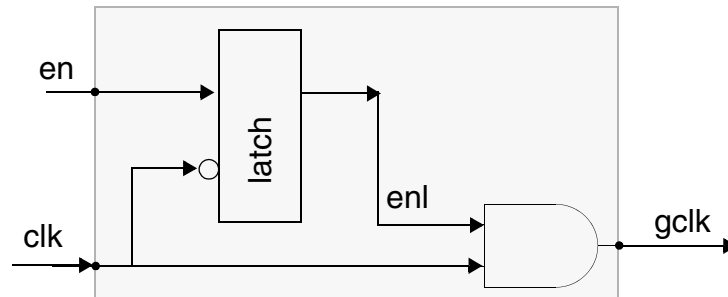
This section contains example schematics of latch-based and latch-free clock-gating styles for rising- and falling-edge-triggered logic. These are a subset of integrated clock-gating cells supported by Power Compiler.

Rising-Edge Latch-Based Integrated Cells

The following integrated cells are latch-based. The rising-edge latch-free integrated cells are described in the following section.

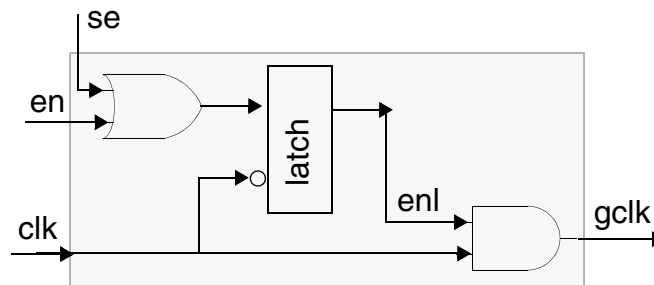
[Figure A-1](#) displays an integrated cell using a latch-based gating style, appropriate for registers inferred from rising-edge-triggered HDL constructs.

Figure A-1 Rising-Edge Latch-Based Integrated Cell (latch_posedge)



[Figure A-2](#) displays an integrated cell using a latch-based gating style, appropriate for registers inferred from rising-edge-triggered HDL constructs. The integrated cell contains test logic (scan enable).

Figure A-2 Rising-Edge Latch-Based Integrated Cell With Pre-Control (latch_posedge_precontrol)



[Figure A-3](#) displays an integrated cell using a latch-based gating style, appropriate for registers inferred from rising-edge-triggered HDL constructs. The integrated cell contains test logic (scan enable).

Figure A-3 Rising-Edge Latch-Based Integrated Cell With Post-Control
(*latch_posedge_postcontrol*)

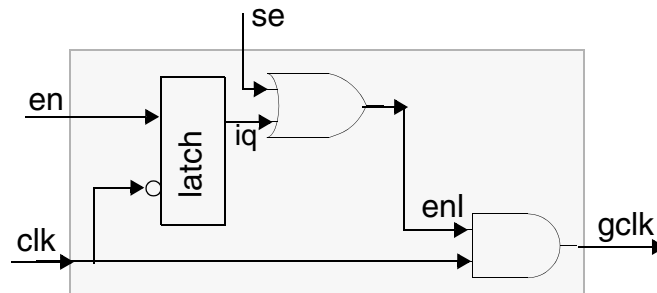


Figure A-4 displays an integrated cell using a latch-based gating style, appropriate for registers inferred from rising-edge-triggered HDL constructs. The integrated cell contains test logic (scan enable) and observable point (cgobs).

Figure A-4 Rising-Edge Latch-Based Integrated Cell With Pre-Control Observable Point
(*latch_posedge_precontrol_obs*)

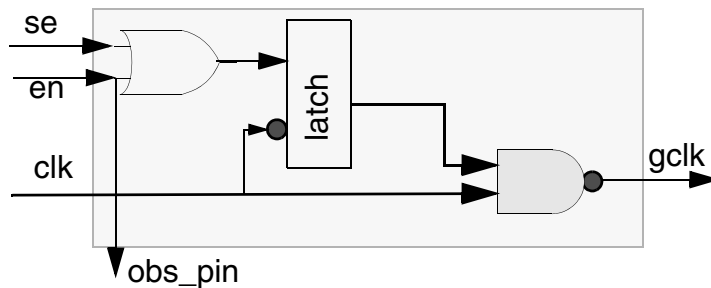
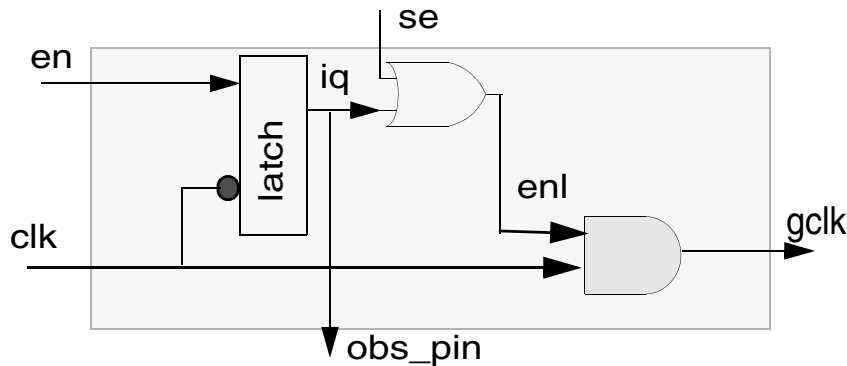


Figure A-5 displays an integrated cell using a latch-based gating style, appropriate for registers inferred from rising-edge-triggered HDL constructs. The integrated cell contains test logic (scan enable) and observable point (cgobs).

*Figure A-5 Rising-Edge Latch-Based Integrated Cell With Post-Control Observable Point
(latch_posedge_postcontrol_obs)*

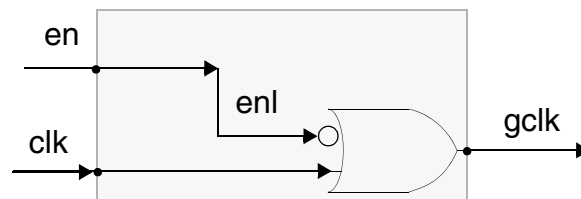


Rising-Edge Latch-Free Integrated Cells

The following integrated cells are latch-free. The rising-edge latch-based integrated cells were described in the previous section.

[Figure A-6](#) displays an integrated cell using a latch-free gating style, appropriate for registers inferred from rising-edge-triggered HDL constructs.

Figure A-6 Rising-Edge Latch-Free Integrated Cell (none_posedge)



[Figure A-7](#) displays an integrated cell using a latch-free gating style, appropriate for registers inferred from rising-edge-triggered HDL constructs. The integrated cell contains test logic (scan enable).

Figure A-7 Rising-Edge Latch-Free Integrated Cell With Control (*none_posedge_control*)

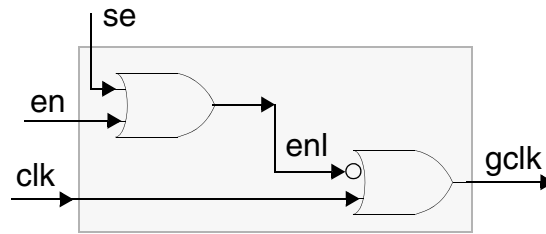
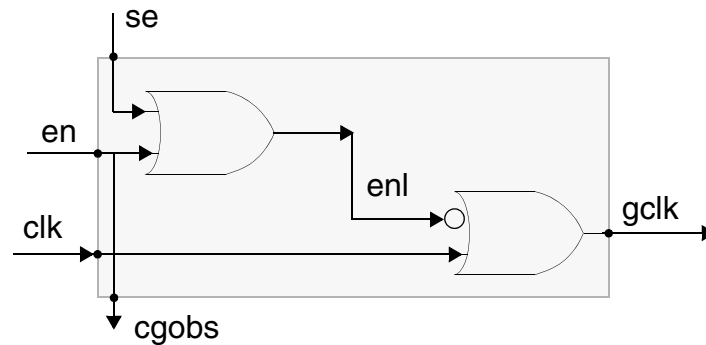


Figure A-8 displays an integrated cell using a latch-free gating style, appropriate for registers inferred from rising-edge-triggered HDL constructs. The integrated cell contains test logic (scan enable) and observable point (cgobs).

Figure A-8 Rising-Edge Latch-Free Integrated Cell With Control Observable Point (*none_posedge_control_obs*)



Falling Edge Latch-Based Integrated Cells

The following integrated cells are latch-based. The falling-edge latch-free integrated cells are described in the following section.

Figure A-9 displays an integrated cell using a latch-based gating style, appropriate for registers inferred from falling-edge-triggered HDL constructs.

Figure A-9 Falling-Edge Latch-Based Integrated Cell (*latch_negedge*)

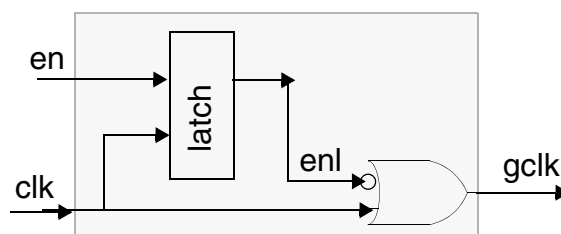


Figure A-10 displays an integrated cell using a latch-based gating style, appropriate for registers inferred from falling-edge-triggered HDL constructs. The integrated cell contains test logic (scan enable).

Figure A-10 Falling-Edge Latch-Based Integrated Cell With Pre-Control Observable Point (*latch_negedge_precontrol*)

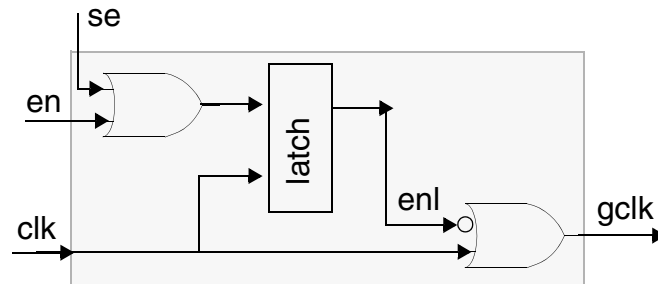


Figure A-11 displays an integrated cell using a latch-based gating style, appropriate for registers inferred from falling-edge-triggered HDL constructs. The integrated cell contains test logic (scan enable).

Figure A-11 Falling-Edge Latch-Based Integrated Cell With Post-Control Observable Point (*latch_negedge_postcontrol*)

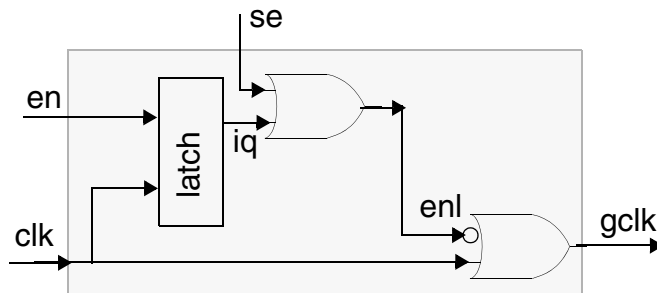


Figure A-12 displays an integrated cell using a latch-based gating style, appropriate for registers inferred from falling-edge-triggered HDL constructs. The integrated cell contains test logic (scan enable) and observable point (cgobs).

Figure A-12 *Falling-Edge Latch-Based Integrated Cell With Pre-Control Observable Point
(latch_negedge_precontrol_obs)*

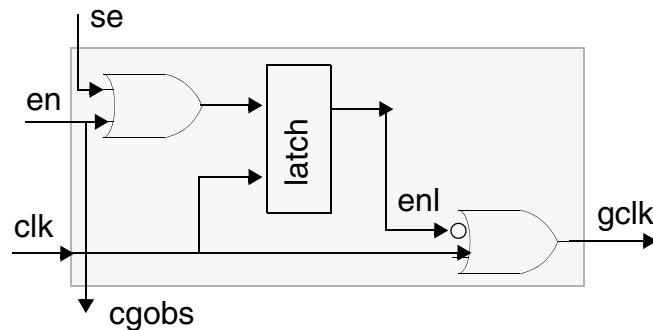
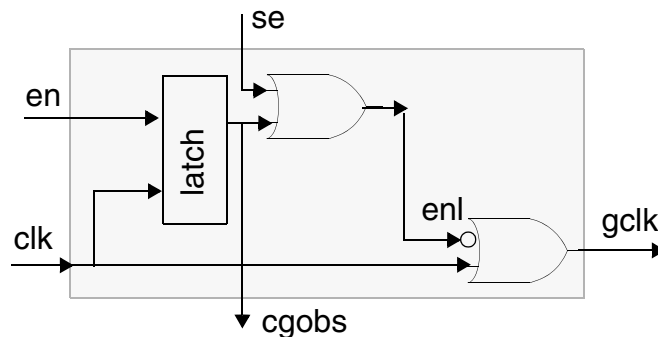


Figure A-13 displays an integrated cell using a latch-based gating style, appropriate for registers inferred from falling-edge-triggered HDL constructs. The integrated cell contains test logic (scan enable) and observable point (cgobs).

Figure A-13 *Falling-Edge Latch-Based Integrated Cell With Post-Control Observable Point
(latch_negedge_postcontrol_obs)*



Falling-Edge Latch-Free Integrated Cells

The following integrated cells are latch-free. The falling-edge latch-based integrated cells were described in the previous section.

Figure A-14 displays an integrated cell using a latch-free gating style, appropriate for registers inferred from falling-edge-triggered HDL constructs.

Figure A-14 Falling-Edge Latch-Free Integrated Cell (none_negedge)

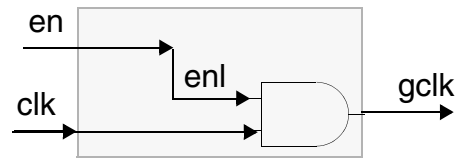


Figure A-15 displays an integrated cell using a latch-free gating style, appropriate for registers inferred from falling-edge-triggered HDL constructs. The integrated cell contains test logic (scan enable).

Figure A-15 Falling-Edge Latch-Free Integrated Cell With Control (none_negedge_control)

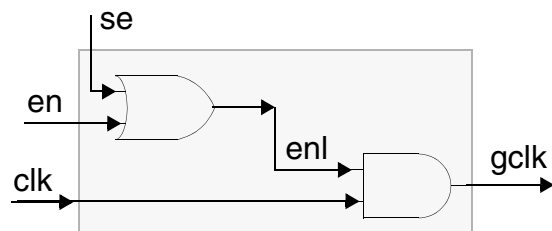
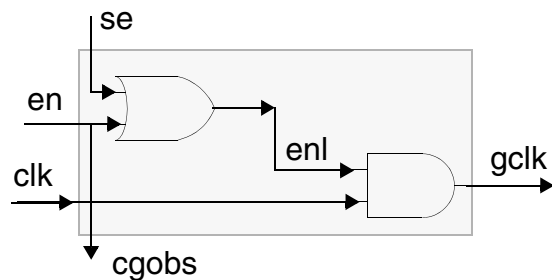


Figure A-16 displays an integrated cell using a latch-free gating style, appropriate for registers inferred from falling-edge-triggered HDL constructs. The integrated cell contains test logic (scan enable) and observable point (cgobs).

Figure A-16 Falling-Edge Latch-Free Integrated Cell With Control Observable Point (none_negedge_control_obs)



B

Attributes for Querying and Filtering

This appendix describes derived Power Compiler attributes that you can use in scripts to view and filter design objects related to clock gating and operand isolation for power optimization.

The derived attributes described in this appendix are read-only properties that Power Compiler automatically assigns to designs, cell, and pins based on other attributes or the netlist configuration.

At times, you may want to view and use design objects according to their attributes. For example, you may want to filter for cells that are integrated clock gates (the `is_icg` attribute). Or, your queries might be required for back end processes such as clock-tree synthesis in which fanout considerations have priority.

This appendix contains the following sections:

- [Derived Attribute Lists](#)
- [Usage Examples](#)

Derived Attribute Lists

You can query for the following derived attributes assigned by Power Compiler. Specify `man power_attributes` in `dc_shell` to view a list of these attributes.

Table B-1 Derived Attributes for Designs

Name	Type	Description
<code>is_clock_gating_design</code>	Boolean	true if the design is a clock-gating design
<code>is_clock_gating_observability_design</code>	Boolean	true if the design is a clock-gating observable design

Table B-2 Derived Attributes for Cells

Name	Type	Description
<code>is_clock_gate</code>	Boolean	true if the cell is a clock gate
<code>is_icg</code>	Boolean	true if the cell is an integrated clock gate
<code>is_gicg</code>	Boolean	true if the cell is a generic integrated clock gate
<code>is_latch_based_clock_gate</code>	Boolean	true if the cell is a latch-based clock-gating cell
<code>is_latch_free_clock_gate</code>	Boolean	true if the cell is a latch-free clock-gating cell
<code>is_positive_edge_clock_gate</code>	Boolean	true if the cell is a positive edge clock gate
<code>is_negative_edge_clock_gate</code>	Boolean	true if the cell is a negative edge clock gate
<code>clock_gate_has_precontrol</code>	Boolean	true if the cell is a clock gate with (pre-latch) control point
<code>clock_gate_has_postcontrol</code>	true if the cell is a clock gate with (post-latch) control point	boolean

Table B-2 Derived Attributes for Cells (Continued)

Name	Type	Description
clock_gate_has_observation	Boolean	true if the cell is a clock gate with observation point
is_clock_gated	Boolean	true if the cell is a clock-gated register or clock gate
clock_gating_depth	integer	number of clock gates on the clock path to this cell; -1 if not a clock gate or register
clock_gate_level	integer	position in a multistage clock tree: number of clock gates on the longest branch in the fan out of this cell; -1 if not a clock gate
clock_gate_fanout	integer	number of registers and clock gates in the direct fan out of the clock gate; -1 if not a clock gate
clock_gate_register_fanout	integer	number of registers in the direct fan out of the clock gate; -1 if not a clock gate
clock_gate_multi_stage_fanout	integer	number of clock gates in the direct fan out of the clock gate; -1 if not a clock gate
clock_gate_transitive_register_fanout	integer	number of registers in the transitive fan out of the clock gate; -1 if not a clock gate
clock_gate_module_fanout	integer	number of modules in the local fan out of the clock gate; -1 if not a clock gate
is_operand_isolator	Boolean	true if the cell is an operand isolation cell
is_isolated_operator	Boolean	true if the cell is an operator that was isolated with operand isolation
operand_isolation_style	string	operand isolation style of the operand isolation cell of isolated operator

For hierarchical clock-gating cells, the derived clock-gating attributes only work when applied to the hierarchical clock-gate wrapper. If you apply an attribute to the leaf cell of a discrete clock gate or a leaf integrated clock gate, the attribute returns false for boolean

attributes, -1 for integer attributes, or an empty string for string attributes. The only exception to this rule is the `is_icg` attribute; this attribute is true when applied to a leaf integrated clock gate contained within a hierarchical clock gate wrapper but false when applied to that wrapper. This behavior allows you to recognize the actual integrated clock-gating cell, not the hierarchical wrapper.

Table B-3 *Derived Attributes for Pins*

Name	Type	Description
<code>is_clock_gate_enable_pin</code>	Boolean	true if the pin is a clock-gate enable input
<code>is_clock_gate_clock_pin</code>	Boolean	true if the pin is a clock-gate clock input
<code>is_clock_gate_output_pin</code>	Boolean	true if the pin is a clock-gate gated-clock output
<code>is_clock_gate_test_pin</code>	Boolean	true if the pin is a clock-gate scan-enable or test-mode input
<code>is_clock_gate_observation_pin</code>	Boolean	true if the pin is a clock-gate observation point
<code>is_operand_isolation_control_pin</code>	Boolean	true if the pin is the control pin of an operand isolation cell
<code>is_operand_isolation_data_pin</code>	Boolean	true if the pin is the data input of an operand isolation cell
<code>is_operand_isolation_output_pin</code>	Boolean	true if the pin is the data output of an operand isolation cell

Usage Examples

You can query the attributes described in the previous section using the `get_attribute`, `get_designs`, `get_cells`, `get_pins`, `all_clock_gates`, and `all_operand_isolators` commands. You can also use these commands with the `-filter` option.

The following examples show how the attributes might appear in scripts.

To gather all the clock gates specific to a clock “clk”:

```
all_clock_gates -clock [ get_clocks clk]
```


The `all_clock_gates` command creates a collection of clock-gating cells or pins that satisfy the parameters you set. Additional options allow you to filter for enable, clock, and gated-clock pins; scan_enable or test_mode pins; and observation pins. For more information, see the man page.

Similarly, the `all_operand_isolators` command creates a collection of operand isolation cells or pins.

To filter out the multistage clock-gating cell associated with the clock “clk”:

```
set multi_stage_cg [filter [all_clock_gates -clock \
    [get_clocks clk]] \ "@clock_gate_level >0" ]
```

To retrieve the number of fan outs of a clock-gating cell:

```
get_attribute [ get_cells top/clk_gate_1 ] \
    clock_gate_fanout
```

To gather a collection of clock-gating cells with pre-latch control point and a fanout greater than four:

```
set CG_collection [filter [all_clock_gates] \
    "@clock_gate_has_precontrol== \
    ==true && @clock_gate_fanout > 4"]
```

To gather a collection of clock-gating designs (the wrapper design where the clock-gating cells reside):

```
set CG_designs [get_designs -filter \
    "@is_clock_gating_design==true"]
```

To gather a collection of operand isolation cells:

```
all_OI_isolators [all_operand_isolators]
```

To query the isolator’s operand isolation style:

```
get_attribute [get_cell C9] operand_isolation_style
```


C

Implementing Multivoltage Designs Using RTL Isolation and Power Constructs

This appendix describes how to implement multivoltage designs using RTL isolation and power constructs as they are coded in Verilog and VHDL. Examples are provided to illustrate how to use the coding rules involved.

This appendix contains the following sections:

- [Using Power Domains](#)
- [Inserting Buffer-Type Level-Shifter Cells](#)
- [Inserting Isolation Cells and Enable-Type Level Shifters](#)
- [Checking the Design for Level Shifters and Level-Shifter Violations](#)
- [Removing Level Shifters](#)
- [Reporting Multivoltage Designs](#)
- [Top-Down Compile Flow](#)
- [Bottom-Up Compile Flow](#)
- [Automated Chip Synthesis Flow](#)
- [Power Compiler Flows for Multivoltage Designs](#)
- [Multivoltage Elements: \\$isolate and \\$power](#)

- [Multivoltage Elements: isolate\(\) and power\(\)](#)

Using Power Domains

Low-power designs are implemented with power domains to control the power consumption of blocks or groups of blocks by selectively switching power on and off to the blocks. Power domains can be independently switched or sequenced through relative always-on relationships.

Power domains are defined hierarchically. When a block is assigned to a power domain, all the subhierarchies are assigned to that power domain, unless a lower-level block is assigned to a different power domain. In this case, all subhierarchies of the lower-level block belong to the newly assigned power domain.

A design contains one or more power domains that are never shut down. By default, such power domains (always-on domains) are mapped with normal cells.

A design can also contain one or more power domains that can be shut down independent of the state of any other power domain. These shut-down domains are specified by using the `-power_down` option when defining the power domain.

In addition, the designs usually contain “*relative* always-on” power domains (defined by the `set_relative_always_on` command). This type of shut-down power domain is required to be on when some other, specified power domain is on; that is, it is on relative to the second power domain. In this way, power can be sequenced through a series of power domains. A relative always-on shut-down power domain can be powered down as long as the other domain is off.

Control signal paths must remain powered up even when a block is powered down. The tool automatically marks these paths as `dont_touch`. To mark these paths manually, use the `get_always_on_logic` command with the `-nets` option to identify the appropriate net and net segments, and then mark them as `dont_touch`.

The cells of the shut-down block that are on a control signal path must remain on during the powered-down phase. This can be achieved by using special, dual-power cells that are appropriately marked as always-on or by using standard, single-power cells that are placed in special always-on site rows within the shut-down block’s voltage area. You use the `set_always_on_strategy` command to specify either the single-power or dual-power strategy for a given power domain.

Note that you have the capability to define and connect multiple power and ground nets for each power domain.

Inferring Power Domains From an RTL Design

Power domains can be defined in the RTL design. In Verilog, the `$power` system task is used, and in VHDL, a power procedure is used. An RTL power domain is essentially defined by the following:

- Power domain name
- Power-down and acknowledge signals (if any)
- Sense information about the power-down and acknowledge signals
- List of modules comprising the power domain

Use the `infer_power_domain` command to translate RTL power domain constructs to power domain objects of the design read by Design Compiler. If you use the `-verbose` option, the equivalent `dc_shell create_power_domain` commands are printed as the power domains are inferred. (See [“Creating Power Domains” on page C-4](#) for the `dc_shell` commands.)

Note:

The `infer_power_domain` command ignores any sense information specified on the power-down and acknowledge signals of the RTL power domain construct.

Creating Power Domains

In the logic synthesis tools, power domains are created with the `create_power_domain` command. To associate a voltage area with a power domain, you specify the voltage area name in this command. This name must be the same as the power domain name.

You can define a given power domain as a power-down power domain, along with specifying the power-down control and acknowledge signals.

Note:

The top-level power domain must be created first.

The `create_power_net_info` command is used to identify both power and ground nets, independent of the power domains. You connect specific power and ground nets to the power pins of the cells of a power domain by using the `connect_power_domain` command.

All cells of the power domain inherit these power and ground connections, unless you specify an exceptional connection. For these cases, use the `connect_power_net_info` command to explicitly connect power and ground nets to the power pins of one or more leaf cells. Exceptional power and ground connections supersede the domain-wise power and ground connections.

Note:

You can use the `report_power_pin_info` command to obtain power pin information about the technology library cells or the leaf cells of the design.

Using the `create_power_domain` Command

The `create_power_domain` command, used to create power domains, includes the following arguments:

Argument	Description
<code>domain_name</code>	Name of the power domain to be created.
<code>-power_down</code>	Optional. Indicates whether this is a power-down domain. If not used, this domain is always-on.
<code>-power_down_ctrl</code>	Optional. Used to specify the power-down control net. Requires the <code>-power_down</code> option.
<code>-power_down_ack</code>	Optional. Used to specify the acknowledge control net. Requires the <code>-power_down</code> option.
<code>-object_list</code>	Optional. Used to list the hierarchical cells associated with the power domain. If not used, this power domain is assumed to be the top-level domain. There can be only one top-level domain. Also, each listed hierarchical cell can belong to only one power domain.

Note:

The `create_power_domain` command does *not* require a uniquified design before using the command. However, if you use the `insert_level_shifters` command in the logic synthesis flow, you must first run `uniquify`.

Other commands directly relating to power domains include

- `get_power_domains` – Use this command to obtain a collection of the power domains defined in the design.
- `report_power_domain` – Use this command to obtain a report on specified power domains.
- `remove_power_domain` – Use this command to delete one or more power domains from the design. You must remove all power domains of the hierarchies before you remove the top-level power domain.

Using the `create_power_net_info` Command

The `create_power_net_info` command, used to specify power or ground net objects, includes the following arguments:

Argument	Description
<code>power_net_name</code>	Name of the power net information object to be created.
<code>-power</code>	Used to specify the power net. Cannot be used with the <code>-gnd</code> option.
<code>-gnd</code>	Used to specify the ground net. Cannot be used with the <code>-power</code> option.

Other commands directly relating to power net information include

- `report_power_net_info` – Use this command to obtain a report on all the power net information objects of the design.
- `remove_power_net_info` – Use this command to delete one or more power net information objects from the design.

Connecting Power Domains

The `connect_power_domain` command, used to connect power and ground nets to power domains, includes the following arguments:

Argument	Description
<code>object_list</code>	List of power domains to be connected.
<code>-primary_power_net</code>	Optional. Name of the primary power net.
<code>-primary_ground_net</code>	Optional. Name of the primary ground net.
<code>-backup_power_net</code>	Optional. Name of the backup power net.
<code>-backup_ground_net</code>	Optional. Name of the backup ground net.
<code>-internal_power_net</code>	Optional. Name of the internal power net.
<code>-internal_ground_net</code>	Optional. Name of the internal ground net.

This command makes domain-wise power and ground net connections. All cells of a power domain inherit these connections, unless an exceptional power connection is specified for a cell or cells (see the next section).

Specifying Exceptional Power Net Connections

The `connect_power_net_info` command can be used to specify exceptional power net hookups to the power pins of certain leaf cells of a power domain. This command requires that these leaf cell power pins are described in the Power-Ground pin format. The command includes the following arguments:

Argument	Description
<code>object_list</code>	List of leaf cells for which exceptional power net connections are made.
<code>-power_pin_name</code>	Name of the leaf cell power pin.
<code>-power_net_name</code>	Name of the power net.

Run the `disconnect_power_net_info` command when you want to remove the exceptional power net connections from a set of leaf cells. After the exceptional connections are removed, the leaf cells inherit the domain-wise power net hookups.

Reporting Power Pin Information

Use the `report_power_pin_information` command to report the power pin information for the technology library cells or the leaf cells of the current design. This command takes only an object list argument of technology library cells or leaf cells.

When you specify a list of library cells, the name, type (power or ground), and voltage specification of the power pin are reported. When you specify a list of leaf cells, in addition to this information, the power net connections are also reported.

Defining Relative Always-On Power Domains

If a power domain should not be shut down when another power domain is on, the former power domain is said to be “always-on” relative to the latter power domain. You establish these relative always-on relationships between power domains by using the `set_relative_always_on` command; it includes the following arguments:

Argument	Description
<code>domain_name</code>	Name of the power domain that remains on relative to other power domains.
<code>-relative_to</code>	List of power domains relative to which a specified power domain is an always-on domain.

The tool uses these relative always-on relationships to determine whether isolation cells are required for nets that cross power domains. Nets with redundant isolation cells are flagged with a warning, and nets with missing isolation cells are flagged as errors.

Finding Always-On Paths

After you have marked the appropriate library and leaf cell pins with the `always-on` attribute, *the tool determines the always-on paths as part of the compile process* (`compile_ultra` or `compile` command). You can use the `get_always_on_logic` command to find the cells and nets of the always-on paths. This command includes the following arguments:

Argument	Description
<code>-cells</code>	Optional. Use this argument to list the cells of the always-on paths.
<code>-nets</code>	Optional. Use this argument to list the nets of the always-on paths.
<code>-all</code>	Optional. Use this argument to list both cells and nets of the always-on paths. Specifying no argument is the same as specifying <code>-all</code> .

For more information about all the commands discussed here relating to power domains, see the appropriate man pages.

Defining a Macro Cell As a Power Domain

A macro cell can be defined as a power domain. Both the `create_power_domain` and `infer_power_domain` commands accept macro cells. The tool automatically recognizes macro cells, so no explicit method of identification is necessary.

However, since a library macro cell does not describe any internal logic, the tool cannot determine whether a given macro cell is internally isolated or requires external isolation. Therefore, you must indicate which case applies to the cell. Macro cells that are isolated internally should have the `is_isolated` attribute set on the appropriate library pin. This attribute can be hard coded in the library or you can set it by using the `set_attribute` command. If a macro cell is not internally isolated, the tool assumes external isolation is required and adds an isolation cell to the appropriate input net.

Note:

The tool honors the `is_isolated` attribute only when it is set on a library pin. If the attribute is set on an instance pin, it is ignored.

Specifying the Voltages of Power Rails

You can directly specify the voltages of the power rails (power nets) of a multivoltage design by using the `set_voltage` command, instead of depending solely on the PVT operating environment factors. The tool uses these layered voltage settings on the power nets as one of the factors in determining the design-intended operating environment. The parts of the design powered by these power nets are timed and optimized at the specified voltages.

If you do not set the voltage on a power net with the `set_voltage` command, the tool uses the available operating condition to determine the voltage. Use the `report_power_net_info` command to see the operating voltages of the power nets.

To change a voltage setting on a power net, you must use the `set_voltage` command again. To clear the voltage settings, use the `reset_design` command.

The command syntax is

```
set_voltage
    max_case_voltage
    [-min min_case_voltage]
    -object_list list_of_power_nets
```

where

- `-max_case_voltage` specifies the voltage that corresponds to the worst case delay (usually the maximum delay).
- `-min min_case_voltage` specifies the voltage that corresponds to the best case delay (usually the minimum delay).
- `-object_list` specifies a list of power nets on which the voltage is set.

A typical flow might be the following:

1. Create a power domain for each unique voltage or shut-down domain.
2. Create a power net for each unique power supply.
3. Use the `set_operating_condition` command only on the design (avoid using this command on block instances).
4. Use the `set_voltage` command to set voltages on the power nets.

Setting an Always-On Strategy

Certain signal paths and cells inside a power-down power domain need to remain powered up when the domain is shut down. The enable signals of isolation cells and enable-type level shifters, as well as the sleep control signals of switch cells and retention registers are such examples. The paths for these signals are known as always-on paths and the cells on these paths are referred to as always-on cells.

There are two methods the tool can use to map always-on cells when synthesizing the always-on paths of a power domain, depending on the strategy you set with the `set_always_on_strategy` command:

- **Dual-power strategy**, which uses special, always-on cells with backup power
 These cells have two power pins – a primary pin that is hooked up to the power domain’s primary power, and a backup pin that is hooked up to the power domain’s backup power. In each logic synthesis session, you must use the `set_attribute` command to mark these cells and their control pins with `always_on` attributes.
- **Single-power strategy**, which uses standard cells used as always-on cells
 These are standard, single-power cells that are placed in dedicated always-on site rows of the power domain’s corresponding voltage area. These always-on sites are powered by the backup power supply. During optimization, the logic synthesis tool does not “know” about these special always-on site rows, so it leaves regular cells on the always-on paths. However, during placement in IC Compiler, the always-on condition is established by constraining these cells to be placed in special, always-on site rows.

The command syntax is

```
set_always_on_strategy
    -object_list {list_of_power_domain_names}
    -cell_type single_power | dual_power
```

You can mix always-on strategies so that some power domains use single-power standard cells and others use dual-power special cells.

To allow certain types of sequential elements to be mapped to specific types or styles of retention registers from the target library during `compile_ultra` or `compile` command, set the `power_enable_power_gating` variable to `true`. The `power_gating_cell` attribute on the library cells, in the target library defines the retention styles of the library cells.

Note:

If you intend to use only dual-power cells, you do not need to use the `set_always_on_strategy` command, because the dual-power strategy is the default behavior.

Optimizing Always-On Logic

Based on pin level always-on attributes, the tool automatically infers the always-on logic. The tool performs timing and area optimization on the always-on logic. At the same time, it makes sure that any illegal logic manually instantiated on the always-on paths is legal. These optimizations are done based the always-on strategy you specify for the power domains.

Constraining the Boundary Nets of a Power Domain

The tool constrains certain boundary nets by marking them `dont_touch`. This is done to prevent

- Buffering a net that would result in a dead cell driving a live cell
- Isolation cells from becoming core cells in the design

Domain-crossing nets divide into the following categories:

- Nets isolated at the driver pins
- Nets isolated at the load pins
- Nets not requiring isolation

Nets Isolated at the Driver Pins

The following rules apply:

- Nets inside the driving power domain are marked `dont_touch`
- Nets inside the load power domain are never marked `dont_touch`
- Nets in always-on power domains are never marked `dont_touch`
- All nets in power-down power domains are marked `dont_touch`

Nets Isolated at the Load Pins

The following rules apply:

- Nets segments inside the driver power domain are never marked `dont_touch`
- All other segments along a route (the pin to pin path) are marked `dont_touch`

Nets Without Isolation

Nets without isolation cells can be constrained with the `dont_touch` attribute only if there's a relative always-on relationship between the power domains. Since a driver power domain is "more always-on" than a load power domain, the route from the driver pin in the "more always-on" domain to the load pin of the second power domain is not constrained.

For multifanout nets, each net is constrained or not constrained independently. If a net is on multiple routes then that net gets the tightest constraint.

Note the following:

- Nets missing a required isolation cell are not constrained. These nets are flagged as violations.
- Always-on nets are not marked `dont_touch` because they are buffered with always-on logic.
- Control signals (power-down nets and power-acknowledge nets) are marked `dont_touch`. If you do not want this behavior, set the `dont_touch_power_domain_control_nets` to `false`.

Inserting Buffer-Type Level-Shifter Cells

There are two types of level shifters (buffer-type and enable-type), and they are not generally part of the original design description. Therefore, they are inserted during the logic synthesis flow. Very different methods are used to add buffer-type and enable-type level shifters to a design.

For *buffer-type* level shifters, these methods include:

- Automatic insertion, as part of running the `compile_ultra` (or `compile`) command
- Manual insertion, by running the `insert_level_shifters` command, usually before you compile the design

Note:

For either insertion method to work, the target library level-shifter cells must be marked with the `is_level_shifter` attribute. In addition, the enable-type level-shifter cells must have their enable pin marked with the `level_shifter_enable_pin` attribute.

Note:

Enable-type level shifters are not inserted by either of these methods. This type of level shifter can be directly instantiated at the RTL level before logic synthesis, or it can be inserted by using the `insert_isolation_cell` command. (See [“Inserting Isolation Cells and Enable-Type Level Shifters” on page C-16.](#))

The command syntax is

```
insert_level_shifters
    -preserve nets
    -all_clock_nets library_cell_name
    -clock_net clock_name
    -verbose
```

where

- `-preserve` is not required.
- `-all_clock_nets` inserts buffer-type level shifter cells for all the clock nets for which voltage stepping is needed. By default the command does not perform level-shifter insertion for clock nets.
- `-clock_net` inserts buffer-type level-shifter cells on the specified the clock nets if voltage stepping is needed. By default the command does not perform any level-shifter insertion on a clock net. This option and the `-all_clock_nets` option is mutually exclusive.
- `-verbose` displays all the level shifters that are found by the command in the target libraries. It also displays all the possible operating points for which each level shifter could be used.

The `insert_level_shifter` command can be used without specifying any of these arguments.

When buffer-type level shifters are inserted (by either method), the tool marks the inserted level shifter cells as `size_only` and sets a `dont_touch` attribute on the port-to-level-shifter nets. The `compile_ultra` (or `compile`) command can then perform sizing optimization on the level shifters. These cells and nets are not removed by any of the optimization procedures.

Both the `compile_ultra` (or `compile`) and the `insert_level_shifters` commands remove any redundant level shifters already present in the design that were not marked as `dont_touch`. (Redundant level shifters are level shifters that have either mismatched voltages or are no longer required on a net crossing power domains due to design changes.) Previously existing level shifters that are not redundant or have been marked `dont_touch` are preserved.

The tool also checks the rail voltages of any preexisting level shifters. If these level shifters have incorrect rail voltages but are still required, the tool replaces them with the correct cells.

You can use the `insert_level_shifters` command to insert buffer-type level shifters at any stage of the design flow. However, it is generally better to insert them early in the flow.

Note:

If you perform logic synthesis by running Automated Chip Synthesis, the buffer-type level shifters are automatically inserted because the tool writes the command into the script before compiling.

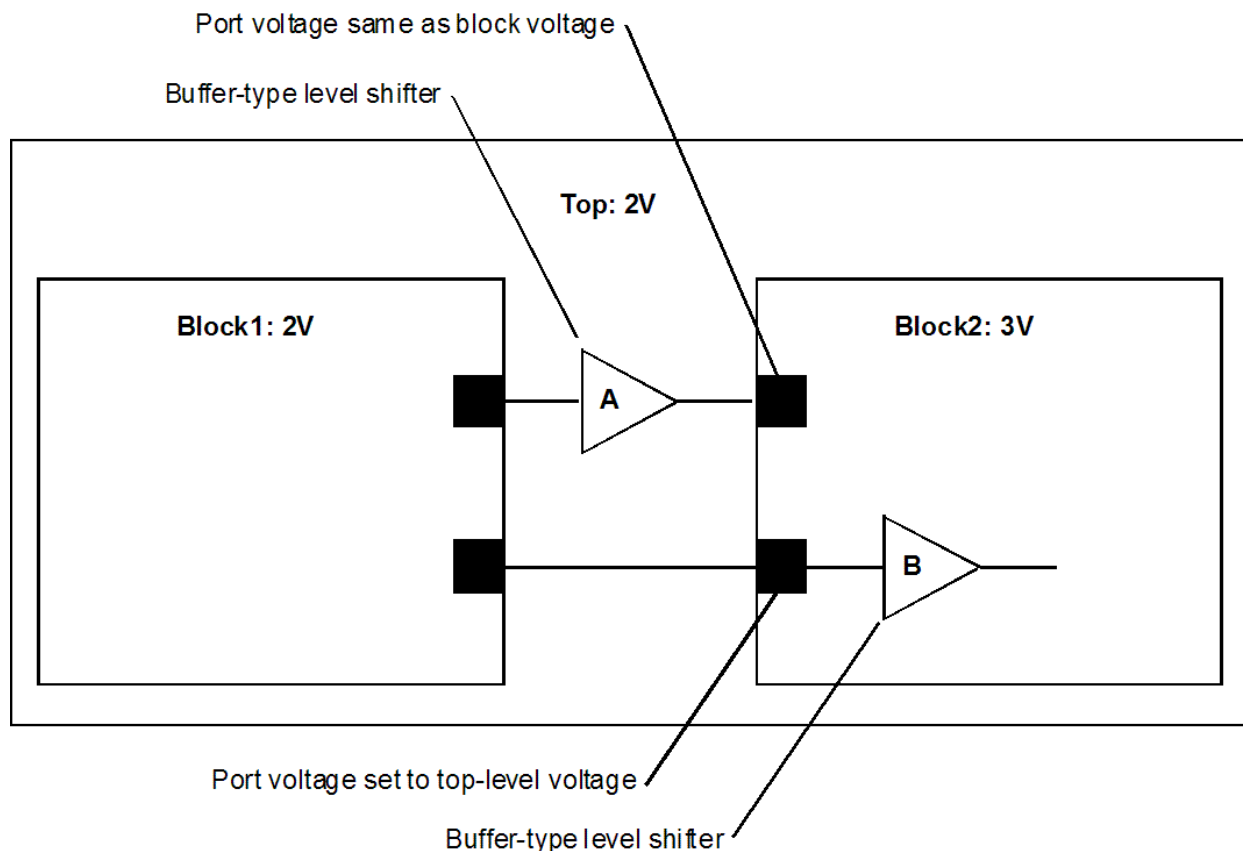
Controlling the Hierarchy Location of Inserted Buffer-Type Level Shifters

In general, buffer-type level shifters are inserted between the blocks, often at the top level of the hierarchy. This is the case when you set the operating voltage on the block instance and not on any of its ports. However, if you set an operating voltage on a block's port that is different from the block operating voltage, the buffer-type level shifter for that port is inserted inside the block.

Specifically, operating voltages are annotated on the instances or their hierarchical ports by using the `set_operating_conditions` command for blocks. Therefore, to insert a buffer-type level shifter inside a block, use the `set_operating_conditions -object_list` command to provide a port list and operating voltage setting for the ports that are different from the block voltage.

Figure C-1 shows an example of buffer-type level-shifter insertion at both the top level for a port that has the default block voltage and within a block for a port that is set at the top-level voltage.

Figure C-1 Buffer-Type Level-Shifter Insertion at the Top Level and Block Level



Controlling the Level-Shifter Strategy and Threshold

When level shifters are inserted, the `compile_ultra` (or `compile`) and `insert_level_shifters` commands use the default level-shifter strategy and the default level-shifter threshold values.

The default level shifter strategy allows the tool to use both step-up level shifters (lower voltage to higher voltage) and step-down level shifters (higher voltage to lower voltage). If you want to override this default, use the `set_level_shifter_strategy` command and specify the `-rule` option as either `low_to_high` or `high_to_low`. (The `all` option is equivalent to the default.)

Note that the default threshold voltage is not zero. Instead, a level shifter is required if any of the following inequalities hold:

- Driver pin $V_{omax} > \text{Load pin } V_{imax}$
- Driver pin $V_{omin} < \text{Load pin } V_{imin}$
- Driver pin $V_{oh} < \text{Load pin } V_{ih}$
- Driver pin $V_{ol} > \text{Load pin } V_{il}$

where the input and output voltage bands of a cell are defined using

- V_{ih} – Lowest input voltage for logic 1
- V_{il} – Highest input voltage for logic 0
- V_{imax} – Maximum input voltage for logic 1
- V_{imin} – Minimum input voltage for logic 0
- V_{oh} – Lowest output voltage for logic 1
- V_{ol} – Highest output voltage for logic 0
- V_{omax} – Maximum output voltage for logic 1
- V_{omin} – Minimum output voltage for logic 0

At least one of these inequalities holds whenever the output voltage band of the driver pin does not completely overlap (is contained within) the input voltage band of the load pin. This criteria applies to both logic 1 and logic 0 voltage bands.

If you want to override the default threshold mechanism, use the `set_level_shifter_threshold` command and specify either an absolute voltage difference (`-voltage` option) or a percentage voltage difference (`-percent` option). The

command sets a minimum voltage threshold for the driver-pin-to-load-pin voltage difference, beyond which a level shifter is required. You can specify both options, in which case a level shifter is inserted if the voltage difference exceeds either or both thresholds.

Note:

If you plan not to use the strategy and threshold defaults, you must set the nondefault values before you run either the `insert_level_shifters` or `compile_ultra` (or `compile`) command.

Inserting Buffer-Type Level Shifters on Clock Nets

The `compile_ultra` (or `compile`) command does not automatically insert buffer-type level shifters on clock nets. If you want the `insert_level_shifters` command to insert buffer-type level shifters on clock nets, you must use the `-all_clock_nets` or `-clock_net` option.

For more information about inserting buffer-type level shifters, see the appropriate man pages for the commands discussed here.

Inserting Isolation Cells and Enable-Type Level Shifters

Isolation cells and enable-type level shifters are used to selectively shut off the input side of the voltage interface of a power domain (see [“Creating Power Domains” on page C-4](#)). Isolation cells do not shift the voltage. Enable-type level shifters usually shift the voltage up or down, but they can also be used as isolation cells in cases where no voltage shift occurs.

Isolation and enable-type level shifter cells can be inserted into the design hierarchy by two methods:

- Manual instantiation at the RTL design description level (isolation cells and enable-type level shifters)
- By using the `insert_isolation_cell` command, which accepts both isolation cells and enable-type level shifters in its cell reference argument

Note:

In the libraries, the `is_isolation_cell` library attribute must be set to `true` for the isolation cells, and the `isolation_cell_enable_pin` attribute must be set on the appropriate input pin of these library cells. Similarly, with enable-type level shifters, the `is_level_shifter` library attribute and the `level_shifter_enable_pin` library attribute must be set appropriately.

When isolation cells or enable-type level shifters are inserted (either method), the tool marks the inserted cell as `size_only` and sets a `dont_touch` attribute on the net that connects the cell to the port.

Manually Instantiating Enable-Type Level Shifters and Isolation Cells at the RTL Design Description Level

An advantage to manually instantiating these cells at the RTL level is that formal verification is possible. For enable-type level-shifter cells, you directly instantiate the cell into the RTL code from the appropriate library. For isolation cells, you can either manually instantiate the cell from the library, or you can use the \$isolate Verilog system task or the VHDL isolate procedure to instantiate a GTECH isolation cell. (For information about these constructs, see [“Multivoltage Elements: \\$isolate and \\$power” on page C-49](#) and [“Multivoltage Elements: isolate\(\) and power\(\)” on page C-51](#).)

The following GTECH cells are available from the GTECH library:

- GTECH_ISO1_EN1
- GTECH_ISO1_EN0
- GTECH_ISO0_EN1
- GTECH_ISO0_EN0
- GTECH_ISOLATCH_EN1
- GTECH_ISOLATCH_EN0

The GTECH isolation cells can be mapped, retargeted, and sized during compile optimization. The hand-instantiated enable-type level shifters can be sized.

Automatically Inserting Isolation and Enable-Type Level Shifters by Using the insert_isolation_cell Command

The `insert_isolation_cell` command allows you to insert isolation cells and enable-type level-shifter cells on specified nets. This command is supported only for designs that contain power domains.

The general intention is that isolation cells are inserted on those nets for which driver and load pins operate at the same voltage, whereas enable-type level shifters are inserted on nets for which the driver and load pins operate at different voltages. Note, however, that enable-type level shifters can be used in place of isolation cells.

The cells that the tool can select for insertion are specified by you through the `-reference` option (see the command syntax that follows). Therefore you can specify enable-type level shifters as well as isolation cells.

This command marks the inserted cell with the `size_only` attribute and the net connecting the cell to the port with a `dont_touch` attribute.

The command syntax is

```
insert_isolation_cell
```

```
-object_list nets
-reference library_cell_name
-enable port/pin/net
-force
```

where

- `-object_list` specifies a list of nets on which isolation or enable-type level-shifter cells can be inserted.
- `-reference` specifies the name of the reference library isolation cell or enable-type level-shifter cell. If a specific library name is not explicitly provided as part of the cell name, all cells from all link libraries matching the cell name are considered.
- `-enable` specifies the name of the signal controlling the enable input of the isolation cell or enable level-shifter cell. If necessary, the tool can route enable signals through the design hierarchies, which can result in the creation of new ports.
- `-force` specifies forced insertion of the isolation cell or enable-type level-shifter cell on `dont_touch` nets.

When inserting an isolation cell or enable-type level-shifter cell, the tool searches all link libraries and selects the first cell it finds that meets all specified multivoltage rules, which include the following:

- The cell matches specified operating conditions.
- The cell is selected from a target library subset if the subset is specified on the net's parent block.
- The cell implements the proper voltage shift or no voltage shift, depending on power domain requirements.

The inserted cell breaks the given net into two parts, requiring names to be assigned to the new nets. The net connected to the data input of the inserted cell retains the name of the original net. The output net is named `ISO_original_net_name`. The naming rule for the inserted cell is `power_domain_name_ISO_#`.

Isolation cells and enable-type level-shifter cells are not inserted on nets with multiple drivers or on nets connected to bidirectional ports.

Note:

Isolation cells and enable-type level-shifter cells change the logic of the original design. This impacts formal verification. The tool issues a message noting the changed functionality.

Use the `remove_isolation_cell -object_list cells` command to remove inserted cells. You can use the `-force` option to remove cells that are marked `dont_touch`.

Note:

When you run the `compile_ultra` (or `compile`) command, the tool sometimes replaces the isolation cells with enable-type level shifters if it can find suitable enable-type level-shifter cells in the target library.

There are a number of ways in which isolation cells and enable-type level shifters might be connected to the input and output nets of the voltage areas (power domains). It is possible to accidentally introduce redundant isolation cells or enable-type level shifters in the RTL description of the design or to fail to specify necessary cells. You can use the `check_mv_design -isolation` command to check for these conditions. (See [“Checking the Design for Level Shifters and Level-Shifter Violations”](#) on page C-19.)

Checking the Design for Level Shifters and Level-Shifter Violations

You can check the status of any existing level shifters in the design by using the `check_mv_design` command with the `-level_shifter` option or the `check_level_shifters -verbose` command. These commands check all level shifters and the level-shifter nets against any specified level-shifter strategy and threshold. Note that the commands can check only those cells for which the library attribute `is_level_shifter` is set to `true`.

If you have not used the `set_level_shifter_threshold` command to override the default strategy and threshold defaults, the tool uses the defaults when checking level-shifter status.

In addition to listing all the level shifters in the design, the `check_level_shifter` command also lists the driver and load pins together with their voltages, and flags the following level-shifter cell violations:

- Incorrect operating condition – The level shifter does not have the correct annotated operating condition.
- Unshifted nets – The nets requiring level shifters
- Wrong level shifter – The given level shifter cannot be used to shift the voltage levels as needed.
- Library cell mismatches – The pin and rail voltages do not agree.
- Inconsistent multiple fanin voltages – The fanins of the level shifter are not all at the same voltage level.
- Inconsistent multiple fanout voltages – The fanouts of the level shifter are not all at the same voltage level.
- Not required – The level shifter is not required according to the defined strategy and/or threshold.

For more information, see the man pages.

Removing Level Shifters

You can use the `remove_level_shifters` command to remove buffer-type level shifters. This command does not remove enable-type level shifters. Note that you cannot remove any level shifters that you manually marked `dont_touch`.

For more information, see the appropriate man page.

Power Gating

Power gating is a technique for reducing leakage current in standby mode without affecting the performance of the design during normal operation.

Power savings are obtained by instantiating multithreshold CMOS retention registers that rely on low-threshold voltage transistors for performance during normal operation, and high-threshold voltage, low-leakage transistors for saving the register state during standby mode.

Power Gating Using Retention Registers

To perform the retention register flow follow these steps:

1. Ensure that the target library contains power-gating cells that have specific cell-level and pin-level attributes for retention operation. This information includes the power on/off behavior and register types.
2. Specify the `set_power_gating_style` command to define the type of retention register to use, to specify which registers within the design should and should not be mapped to retention registers, and to specify which HDL block to be mapped.

You must supply a `-type_name`, which by default is applied to all registers throughout the design hierarchy. To map portions of the design to retention registers, specify the desired cells or designs. Additionally, you can apply the `-hdl_blocks` option to map the sequential elements generated by named RTL processes to retention registers. This option applies to the `current_design` by default, unless the `-hier` option is added.

Details about the various use models are covered in the following section.

3. Specify the `set_power_gating_signal` command to set the attributes on the ports or pins to indicate which power-gating pins of which retention register type use those ports/pins to connect to or through the corresponding design hierarchy.

4. Set the `power_enable_one_pass_power_gating` variable to `true` to enable this one-pass retention register flow.
5. Specify `hookup_power_gating_ports` to define port-naming conventions for use during the wiring process and to enable control pin wiring during compile.

For port-naming conventions, specify an ordered list of `pin_names` with the `-port_naming_style` option, or apply a default naming style.

6. Compile.

During compile, Power Compiler synthesizes the applicable registers to retention registers and wires their control pins based on attributes set with the `set_power_gating_signal` command.

Based on your specifications, Power Compiler determines which registers to map to retention registers and checks the library for possible candidates by isolating the cells with the appropriate power-gating cell attributes.

To remap any part of the design into ordinary registers, you must go back to the original netlist and perform a full compile operation.

The following script shows the retention register flow:

```
set_power_gating_style -type CLK_FREE [get_cells_\
    lev1b_inst/*reg*]
set_power_gating_style -type CLK_FREE [get_cells_\
    lev1c_inst/dout2*]
set_power_gating_style -type CLK_FREE [get_cells_\
    lev1c_inst/dout1*]

set_power_gating_signal -power_pin_index 1 [get pin
    lev1b_inst/retain]
set_power_gating_signal -power_pin_index 2 [get pin
    lev1b_inst/shutdown]
set_power_gating_signal -power_pin_index 2 [get pin
    lev1c_inst/retain]
set_power_gating_signal -power_pin_index 1 [get pin
    lev1c_inst/shutdown]

set power_enable_one_pass_power_gating true

hookup_power_gating_ports -port_naming_style {test1 test2}\
    -default_port_naming_at_style

compile_ultra

write -format verilog -hierarchy -output post_compile.v
```

Use the `report_power_gating` command to review the registers that were mapped to retention registers for the specified design. If no design is specified, the `current_design` is assumed. For more information about these commands, see the man pages. The remainder of this appendix describes this process in detail.

Use Models for Mapping

Based on your requirements, the `set_power_gating_style` command can be applied to perform the desired mapping. The following section lists a number of different use models and how to apply the `set_power_gating_style` command to map to the desired retention registers.

Power Gating Default Type

By default, the `-type` option applies to all registers throughout the current design. Where possible, Power Compiler maps sequential elements to retention registers with the specified attribute. It applies to the complete design hierarchy.

If omitted, or incorrectly set, Power Compiler generates an error, and no mapping to retention registers is performed during compilation.

Based on the HDL Block Name

You can map the registers defined in named RTL blocks. Shown below is an example of how to use the `-hdl_blocks` option of the `set_power_gating_style` command to map only the registers within the particular blocks to retention registers.

Figure C-2 RTL Example With a Named Block

```

module reg16c (regin, regout, clk, gate, clr);
input gate;
input clr;
input clk;
input [15:0] regin;
output [15:0] regout;
reg [15:0] regout;
    always @(posedge clk)
    begin: reg16c_proc
        if (clr) regout = #1 16'b0;
        else if (gate) regout = #1 regin;
    end
endmodule

```

Process name is **reg16c_proc**

To map the registers within a process to a particular type of retention register, you must know the available types which are specified in the retention register library cells with the `power_gating_cell` attribute.

Figure C-3 Sample Library Cell

```
Cell (DFF_RT)
,Ä¶,
power_gating_cell : ,Ä¶ret1
,Ä¶
pin (sleep){
,Ä¶
power_gating_pin (power_pin1,
,Ä¶0,Ä¶
pin (wake){
,Ä¶
power_gating_pin (power_pin2,
,Ä¶0,
```

Cell type is **ret1**

The type of retention register to be used is specified with the `set_power_gating_style` command.

Figure C-4 Sample Tcl Script

```
set search_path {. lib1}
set link_library {* lib1.db}
set target_library {lib1.db}
set_power_gating_style ,Ä¶type ret1 ,Ä¶hdl_block reg16c_
set power_enable_one_pass_power_gating true
compile_ultra -scan
write ,Ä¶f verilog ,Ä¶hier ,Ä¶o reg
report_power_gating_style > pwg.rpt
quit
```

Cell type is **ret1** from library attribute

Process name is **reg16c_proc** from RTL

The synthesized netlist includes retention registers for all the registers defined within the **reg16c_proc** process. They are of type “**ret1**”. The sleep and wake signals are connected to their disabled state, which in this case is logic “0”.

Note:

The section on wiring of power-gating pins describes the commands required to connect the power-gating pins to a port or signal instead of the disabled logic value.

Named Blocks per Retention Register Type

The `set_power_gating_style` command supports the use of the wildcard for referencing multiple process blocks within the design. This is useful when naming the processes based on the intended retention register type.

For example, assume the following named processes are in the RTL description:

- `begin: type1_proc_reset`
- `begin: type2_proc_hold`
- `begin: type1_proc_transfer`

A wildcard can be used to assign all the processes with the `type1` prefix to retention registers of `type1`; and all the processes prefixed with `type2` to `type2` as shown with the commands below:

- `set_power_gating_style -type type1 -hdl_block type1*`
- `set_power_gating_style -type type2 -hdl_block type2*`

By default, the `set_power_gating_style` command operates on the current design level. To apply the commands to the complete hierarchy, apply the `-hier` option as shown below:

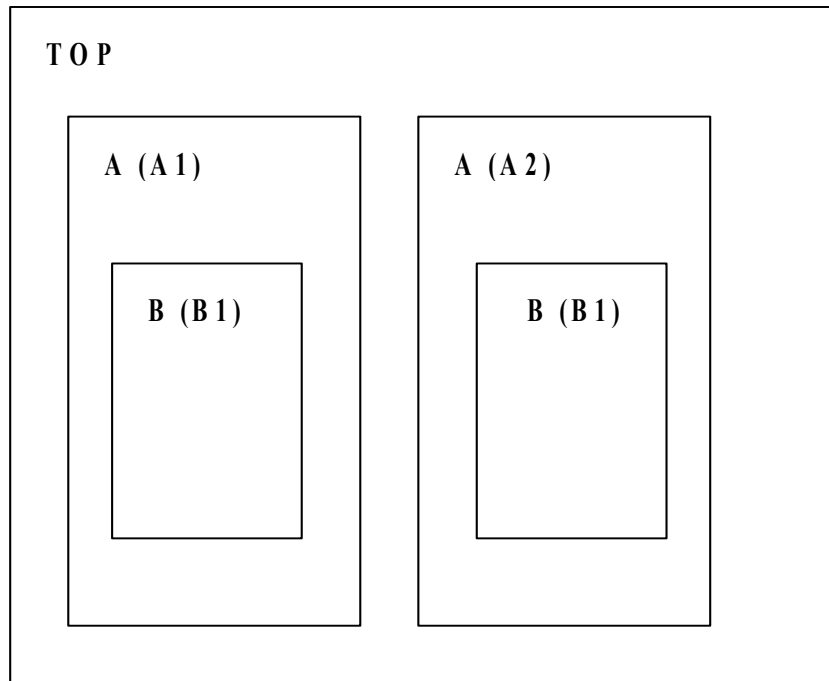
- `set_power_gating_style -hdl_block type1* -type type1 -hier`
- `set_power_gating_style -hdl_block type2* -type type2 -hier`

The `-hier` option applies only when the `-hdl_block` is applied.

Design-Level Mapping

The following sections describe how to use the `set_power_gating_style` command to map designs or subdesigns to particular retention registers by providing examples for various scenarios given the same design: TOP.

Figure C-5 Design TOP



- Applying a power-gating style “FREE” to the entire design TOP

```
current_design top
set_power_gating_style -type FREE
```

Since the `current_instance` is TOP, the power gating style FREE is applied to every instance within the complete design hierarchy. All registers in A1, A2, B1 and B2 are mapped to retention registers of type FREE.

- Applying a power-gating style “FREE” to the module A

As long as there is no default type set to the current instance, you can set the power-gating style for subdesigns of the current design.

```
current_design top
set_power_gating_style -type FREE A
```

With these commands, the registers at the top level are not mapped to retention registers. The registers within instances A1 and A2 are mapped to retention registers of type FREE. The type is applied hierarchically so that all the registers in A1/B1 and A2/B1 are also mapped to retention registers of type FREE.

- Applying a power gating style “FREE” to the module A and “LOW” to module B

As long as there is no default type set to the current instance, you can set the power-gating style for sub-designs of the current design. The low-level subdesign specification has precedence; so the following commands allow you to specify a type for registers within module A and a different type for registers within module B.

```
current_design top
set_power_gating_style -type FREE A
set_power_gating_style -type LOW B
```

With these commands, the registers at the top level are not mapped to retention registers. The registers within instances A1 and A2 are mapped to retention registers of type FREE—although the registers within the sub-design B are mapped to type LOW; for example, A1/B1 and A2/B2 registers are mapped to type LOW.

- Applying a power-gating style “FREE” to the TOP module and “LOW” to module A and “HIGH” to module B

Since the current design is the current instance, perform this mapping on a per instance basis as shown with the commands below:

```
current_design top
set_power_gating_style -type FREE
current_instance A1
set_power_gating_style -type LOW
current_instance ../A2
set_power_gating_style -type LOW
current_instance ../A1/B1s
set_power_gating_style -type HIGH
current_instance ../A2/B1
set_power_gating_style -type HIGH
current_instance ..
```

Registers within TOP are mapped to type FREE, registers within A1 and A2 are mapped to type LOW and the registers within A1/B1 and A2/B1 are mapped to type HIGH.

Note:

You cannot set the power-gating style on sub-designs by changing the `current_design`. Power Compiler applies the top-level `current_design` to the entire hierarchy. Consider the sample commands below:

```
current_design top
set_power_gating_style -type FREE
current_design A
set_power_gating_style -type LOW
current_design B
set_power_gating_style -type HIGH
```

The power-gating style FREE is applied to ALL registers within the complete design hierarchy including instance A1, A1/B1, A2, and A2/B1. Regardless of the order in which these are applied, Power Compiler applies the default type set on the current instance.

Instance-Based Mapping

To perform mapping on instances other than leaf-cells, set the `current_instance` command to the desired instance; and apply the `set_power_gating_style` command. For example; if there are two instances U1 and U2 of module A within the top-level design and you wish to map only the registers in U1 to retention registers, the following command set could be used if the instance is at the RTL level:

```
current_instance U1
set_power_gating_style -type type1
```

If the instance is at the gate level:

```
set_current_instance U1
set_power_gating_style -type type1 [get_cells * -hier]
```

Specification of Which Registers Not to Map

If most of the registers are to be mapped to retention registers, specify which portions of the design should NOT be mapped to retention registers by applying a power gating type “none”. In this example, the registers are not mapped to retention registers within the instance U1.

```
current_instance U1
set_power_gating_style -type none
```

Re-Mapping of Gate-level Designs to Retention Registers

Power Compiler supports re-mapping of gate-level netlist registers to retention registers. Once the netlist is read in, select power gating styles to map the selected registers to retention registers instead. For example, if the netlist contains instances U1_reg[0] and U1_reg[1] of normal flip-flops, you can select to map these to retention registers instead with the command:

```
set_power_gating_style -type type1 {U1_reg[*] }
```

Or, if all sequential elements should be replaced, you can apply the command:

```
set_power_gating_style -type type1
```

In this case, all registers throughout the design hierarchy are replaced if a retention register counterpart is available.

In any incremental compile operation, be sure to use the `-scan` option if that option was used in the original compile operation. Omitting the `-scan` option disconnects and disables the power-gating ports of any retention registers, and it also remaps ordinary scan registers to non-scan registers.

After registers have been mapped to retention registers, it is not possible to remap them back into ordinary registers in an incremental compile operation. To remap any part of the design into ordinary registers, you must go back to the original netlist and perform a full compile operation.

Wiring Power-Gating Pins

During compile, Power Compiler wires the power-gating pins to the appropriate signals, which places the retention cells in normal mode or power-down mode. In power-down mode, the state is saved in the high-threshold voltage balloon latch.

- Power Compiler uses a bottom-up approach to perform the hookup. It wires the ports or pins with compatible attributes until reaches a port of the `current_design` or a driver pin of a cell (meaning that the power-gating signal was generated by internal logic).
- By default, Power Compiler connects all power-gating pins with the same library `pin_name` and `power_gating_pin` attribute values to top-level ports of the design. Where needed, ports are created throughout the hierarchy to connect the retention cell pins ultimately to the top-level port.
- If the design already contains ports for the power gating signals, or if the power gating signals are generated via internal logic, identify these ports/pins and the `power_gating_pin_name` or `power_gating_pin` index value to which they should be connected. You can identify the following power-gating signals:
 - a top-level port
 - a hierarchical pin
- Optionally, connect the power-gating pins based on type.
- Optionally, connect power-gating pins that have different `power_gating_pin` index values. Basically
 - The `power_gating_pin[1-5]` index must be the same.
 - The `power_gating_pin` disabling value must be the same ["0" | "1"].

Specifying Wiring Rules

Use the `set_power_gating_signal` command to define how Power Compiler should wire the retention registers, as follows:

- `set_power_gating_signal <ports/pins> [-library_pin <lib_pin> | -power_pin_index <index>] [-type]`

If the power-gating ports/pins are already created within the design, this command is used to identify them. It can be applied to the following pins/ports.

- Pre-defined pins to the modules with retention cells, which were user-created knowing that the power-gating signals are needed.
- Pre-defined output pins of modules, which generate the power-gating signals.
- Top-level ports which supply the power-gating signals for the retention registers.

If the port/pin does not exist, Power Compiler issues an Error/Warning. Refer to the library `pin_name` or the `power_gating_pin index` value, which ever is more convenient.

To define all the power-gating signals, issue this command multiple times. Power Compiler places the following attributes on the pins and ports listed with the `set_power_gating_signal` command:

- `power_gating_style` (if `type` option was applied)
- `power_pin_class` (`power_pin index 1-5`)

The `-type` option is elective because multiple types of retention cells may be connected to the same power-gating signal.

When applying the attributes, Power Compiler performs the following tasks:

- It verifies the specified port/pin exists in the design.
- It verifies the `<pin name or power_gating_pin index>` exists on the `power_gating_cells` and has the attribute `power_gating_pin`.
- The `power_pin_class` attribute is assigned the `power_gating_pin index` value of the `power_gating_pin`.
- If the `-type` option was applied, it checks that power-gating cells of the type specified are instantiated in the design and add the `power_gating_style` attribute to the specified ports/pins specified.
- If issued multiple times, any conflicts in the `power_gating_style` or `power_pin_class` are flagged. If applied incorrectly, remove the attributes with the `remove_attribute` command, and reissue `set_power_gating_signal`.

Described below are the guidelines applied by Power Compiler for wiring as Power Compiler works its way up through the hierarchy.

- If Power Compiler encounters an output port of a module with the `power_gating_signal` attributes, it stops there. It understands that the module output is a power-gating signal and no top-level port is required. Otherwise, it continues until it gets to the top-level of the current design. If no top-level ports exist with the user-specified hookup options, top-level ports are created based on the naming style.
- If a lower-level port has no `power_gating_style` attribute but the higher-level port does, wiring is not performed.

- Power Compiler does not wire pins with incompatible `power_gating_style` and `power_pin_class` attributes.

Use Models for Wiring

The following sections describe how to perform mapping and wiring for various use models.

Wiring by Power-Gating Cell Type

In general, map portions of the design to various types of retention latches/registers. The following commands can be used to map and wire based on the `set_power_gating_style` “type” for the entire design.

- Mapping by type

```
set_power_gating_style -type ret1 -hdl_block *ret1* -hier
set_power_gating_style -type ret2 -hdl_block *ret2* -hier
set power_enable_one_pass_power_gating true
compile_ultra
report_power_gating_style > pwg_pre.rpt
```

At this point, if you do not know the name, index and disabled value of the `power_gating_pins` for each retention cell type; the power gating report is of value as it lists the power-gating pin names and attribute values for the retention cells in the design. Shown below is a sample output generated by the above commands.

Based on whether the `power_gating_signals` command exist already, or if they need to be created; different command options can be applied to wire the signals based on the type.

- Specifying existing signals for wiring

If pins or ports exist already within the design, then the following commands can be used to identify them as `power_gating_signals` so that they are hooked up to the appropriate pins on the retention cells, based on the type.

```
set_power_gating_signal -type ret1 -library_pin save \
{save_ret1}
set_power_gating_signal -type ret1 -power_pin_index 2 \
{rest_ret1}
set_power_gating_signal -type ret2 -power_pin_index \
{save_ret2}
set_power_gating_signal -type ret2 -library_pin restore \
{rest_ret2}
hookup_power_gating_ports -type ret1 -port_naming_style \
{save_ret1, rest_ret1}
hookup_power_gating_ports -type ret2 \
-default_port_naming_style {save_ret2_%d,,
rest_ret2_%d}
```


The `set_power_gating_signal` command assigns the attributes to identify the existing port as a `power_gating_port`. Notice that you can specify either the library pin name, or the `power_gating_pin_index` when defining the power gating signals. Also, the `default_port_naming_style` can be used to create names similar to those specified with the `port_naming_style`.

- Identifying unconnected retention cells

If you omitted the commands to hookup some of the retention cell power-gating pins, the report would show the disabling value, instead of the port connection.

Wiring by Power-Gating Pin Name

You should be able to connect all power-gating pins with a specific library pin name and `power_gating_pin` attribute values (i.e. `power_gating_pin1`, `power_gating_pin2`..... `power_gating_pin5`) to either design ports or pins within the design. In this case, the connections can be made based on the library `pin_name` regardless of the `power_gating_cell` type. Power Compiler checks that they all have the same attribute values before making the connection.

Consider the case where all the retention cell types have two power-gating pins; for example, `save` and `restore` and have the same disabled value. You may wish to globally connect all pins with the same name and, of course, the same attribute values: `pin_name: save power_gating_pin_index 1 (power_gating_pin1)` to one port; and `pin_name: restore power_gating_pin2` to another port etc.

- Wiring based on the pin name

The following commands can be used to connect an existing port (`alltypes_save`) to one of the `power_gating_pins` (`power_gating_pin1`) and to create new ports for the other `power_gating_pins` (`restore: power_gating_pin2`)

The resulting report file would show the power-gating signals connected as shown below.

Figure C-6 Power-Gating Cell Report

Power Gating Cell Report			
Cell Name (Library Cell Name)	Power Gating Style	Power Gating Pin	Signal
RegB/out_reg[1] (DFF_RET1) ret1	restore (2)	restore	
	save (1)	alltypes_save	
RegB/out_reg[0] (DFF_RET1) ret1	restore (2)	restore	
	save (1)	alltypes_save	
RegB/out1_reg[1] (DFF_RET2) ret2	restore (2)	restore	
	save (1)	alltypes_save	
RegB/out1_reg[0] (DFF_RET2) ret2	restore (2)	restore	
	save (1)	alltypes_save	

Reporting Retention Registers

To verify the wiring is correct and all retention cell power-gating pins are connected, use the `report_power_gating` command. The disabling values are replaced with the actual port connections made. Shown below is a sample report file generated with the `report_power_gating > pwg_post.rpt` command after wiring has been performed.

Shown below is a power-gating report:

Figure C-7 Power-Gating Report

Power Gating Cell Report			
Cell Name (Library Cell Name)	Power Gating Style	Power Gating Pin	Signal
RegB/retout_reg[3] (DFF_RT)	clock_free	REST (2)	restore
		SAVE (1)	save
RegB/retout_reg[2] (DFF_RT)	clock_free	REST (2)	restore
		SAVE (1)	save
RegB/retout_reg[1] (DFF_RT)	clock_free	REST (2)	restore
		SAVE (1)	save
RegB/retout_reg[0] (DFF_RT)	clock_free	REST (2)	restore
		SAVE (1)	save

The report can be used to determine which `power_gating_pins` are specified for each cell, as well as the disabling value to assist them in determining how to connect them.

Reporting Multivoltage Designs

Certain checking and reporting commands are appropriate to use with multivoltage designs. Most of these commands have been enhanced to provide explicit PVT values as well as identifying the given operating conditions.

Enhancements to Report Commands Used With Multivoltage Designs

The operating condition is not adequate in describing and reporting the operating environment of designs containing complex cells. Therefore, the explicit values of the process, voltage, and temperature variables (the PVT variables) and voltage map information (where applicable) are now used to define the operating environment.

In particular, with the report command enhancements, you can

- List the PVT values (as well as the operating conditions) in the reports
- Check for different PVT values set on different blocks
- Determine whether each block operates at its specified PVT
- Check the PVT values of any target library subsets used in the multivoltage design flow

The affected report commands include

- `report_timing` – change: Lib:OC column replaced with a P, V, T column of values
- `report_cell` – changes for single rail cells:
 - Lib:OC column replaced with a P, V, T column of values
 - Rail information removed
- `report_cell` – changes for level shifter cells:
 - Lib:OC column replaced with a P, V1 -> V2, T column of values
 - Two power rails voltages associated with the level shifters reported
 - Rail information removed
- `report_cell` – changes for dual rail cells: primary and backup power information provided
- `report_delay_calculation` – delay calculation from output pin of a level shifter to load input pin reported with these changes:
 - Correct (and precise) pin operating condition provided
 - Operating condition describing the cell's operating environment removed
 - Rail information removed
- `report_hierarchy` – changes include
 - Relevant voltage map reported for dual rail cells
 - Column of P, V, T values reported for single rail cells
 - Lib:OC column removed
- `report_power` – changes include
 - Listed operating conditions removed
 - Operating environment listed for every block
- `report_power_calculation` –
 - Listed operating conditions removed
 - Operating environment listed for every block
- `report_design` – operating condition, library, interconnect model, and P, V, T values still reported but header format changed so that the P, V, T values appear as a separate grouping
- `report_target_library_subset` – change: Opcond column replaced with a P, V, T column of values

Note:

The `report_net` command is not affected because this command does not report operating conditions.

Reports Used With Multivoltage Designs

The following reporting and checking commands are used with multivoltage designs:

report_operating_conditions

This command report displays all operating conditions currently defined in the target and linked libraries as well as any operating conditions you created using the `create_operating_condition` command.

delete_operating_condition

The syntax for this command is

```
delete_operating_condition opcond_name
```

You run this command when you want to remove any operating condition that you created with the `create_operating_condition` command. The user-specified operating condition with the name `opcond_name` is deleted from the set of operating conditions, and the following warning is issued:

```
Warning: Operating condition opcond_name has been deleted.
```

However, if any subdesign uses the `opcond_name` operating condition, the command does not remove this operating condition. Instead the following warning is issued:

```
Warning: Design design_name is using the operating condition  
opcon_name. Please first change the operating condition for  
the design design_name before deleting it. opcon_name is not  
deleted.
```

The `remove_operating _condition` command does not remove library-specified operating conditions.

report_target_library_subset

Use this command to find out the target library constraints, that is, to determine or confirm which target library subsets are assigned to which design instances.

check_mv_design

Use this command to check for design errors, including multivoltage constraint violations, electrical isolation violations, connection rules violations, and operating condition mismatches. Two switches let you control the level of information detail and limit the number

of messages printed to the log file. Other switches, such as `-isolation`, `-level_shifter`, `-connection_rules`, `-opcond_mismatches`, and `-target_library_subset`, let you select among the available checking reports.

If you do not specify any checkers, violations are reported from all supported checkers. If you use the `-verbose` switch, the checkers provide detailed reports. If you omit this switch, summary reports are provided.

This command includes the following arguments:

Argument	Description
<code>-verbose</code>	Optional. Provides a detailed report. If you do not use this option, a summary of any violations is reported.
<code>-max_messages</code> [message count]	Optional. Sets a limit, given by <i>message count</i> , on the number of messages per checker printed in the log file. If no checkers are specified, this is the message limit for all checkers. If you do not use this option, all messages are printed.
<code>-isolation</code>	Optional. Provides a report on electrical isolation errors with respect to power domains.
<code>-level_shifter</code>	Optional. Provides a report on all existing level shifters and connecting nets. Checks against the specified level shifter strategy and threshold.
<code>-connection_rules</code>	Optional. Reports violations in always-on synthesis and pass-gate connections.
<code>-opcond_mismatches</code>	Optional. Reports incompatible operating conditions between instantiated technology cells and the cells' parent design.
<code>-target_library_subset</code>	Optional. Reports inconsistent settings among target libraries, target library subsets, and operating conditions.

check_level_shifters

Use this command to check all level shifters, including hand-instantiated enable-type level shifters, and level-shifter nets against any specified level-shifter strategy and threshold. Note that the command checks only cells for which the `is_level_shifter` library attribute is set to `true`.

check_isolation_cells

Use the `check_isolation_cells` command to check for the presence of isolation cells in the mapped logic hierarchies (blocks) or the voltage areas associated with the power domains of the design. This command reports the number of isolation cells, informs you if any of these cells are redundant, and warns you if isolation cells that might be needed are missing. Using the command options, you can restrict the report to check only for isolation cells that are

- On the input nets of the specified logic hierarchies or voltage areas
- On the output nets of the specified logic hierarchies or voltage areas
- Inside the specified logic hierarchies or voltage areas
- Outside the specified logic hierarchies or voltage areas

Note:

You can also use this command to check for isolation cells in the mapped netlist before you run IC Compiler to place and optimize the design. In this case, you specify the logic hierarchies (blocks) associated with the given power domains instead of the voltage areas.

report_cell

In multivoltage mode, this report provides additional columns to display the PVT values and voltages annotated on the cells of the design.

check_design

In multivoltage mode, this command provides an additional error message: “Error: cell ‘%’ is not characterized for %fV.” For example,

```
Error: cell InstDecode/U3(INVX1) is not characterized for 0.900000V.  
(MV-001)
```

report_timing

In multivoltage mode, this command reports the library operating condition, voltage, and temperature of the cells in the timing paths instead of the design operating condition.

The library operating conditions of the cells in the timing path are displayed under the report column “Lib:OC.” Voltages are displayed under a “Voltage” column, and temperatures under a “Temperature” column (if you use the appropriate options).

report_delay_calculation

In multivoltage mode, the command also reports PVT values, operating conditions, and voltages.

report_hierarchy

In multivoltage mode, the command also reports PVT values, operating conditions, and voltages.

report_power

This command calculates and reports the power usage of both multivoltage and single-voltage designs. The command requires switching activity information for all design nets, from which it calculates and displays net switching power, cell internal power, and cell leakage power. Switching activity information is usually annotated by the user. A switching activity propagation mechanism is used to estimate switching activity information about non-annotated design objects. The PVT values of the blocks are reported.

report_power_calculation

This command displays the calculation of internal power for a pin, leakage power for a cell, or switching power for a net. PVT values are reported.

report_clock_gating

This command reports information about clock gating cells and gated and ungated registers of the current design. Note that it only recognizes clock gating cells for which the clock gating attributes are marked by Power Compiler. PVT values are reported.

report_power_gating

This command reports the power gating style of retention registers. The library retention registers have a cell level attribute `power_gating_cell` to specify the power gating styles of the registers. PVT values are reported.

For more information about these checking and report commands, see the appropriate man pages.

Top-Down Compile Flow

The top-down compile flow is the recommended flow.

You can run a top-down compile on a multivoltage design, using multiple nonlinear delay model (NLDM) libraries.

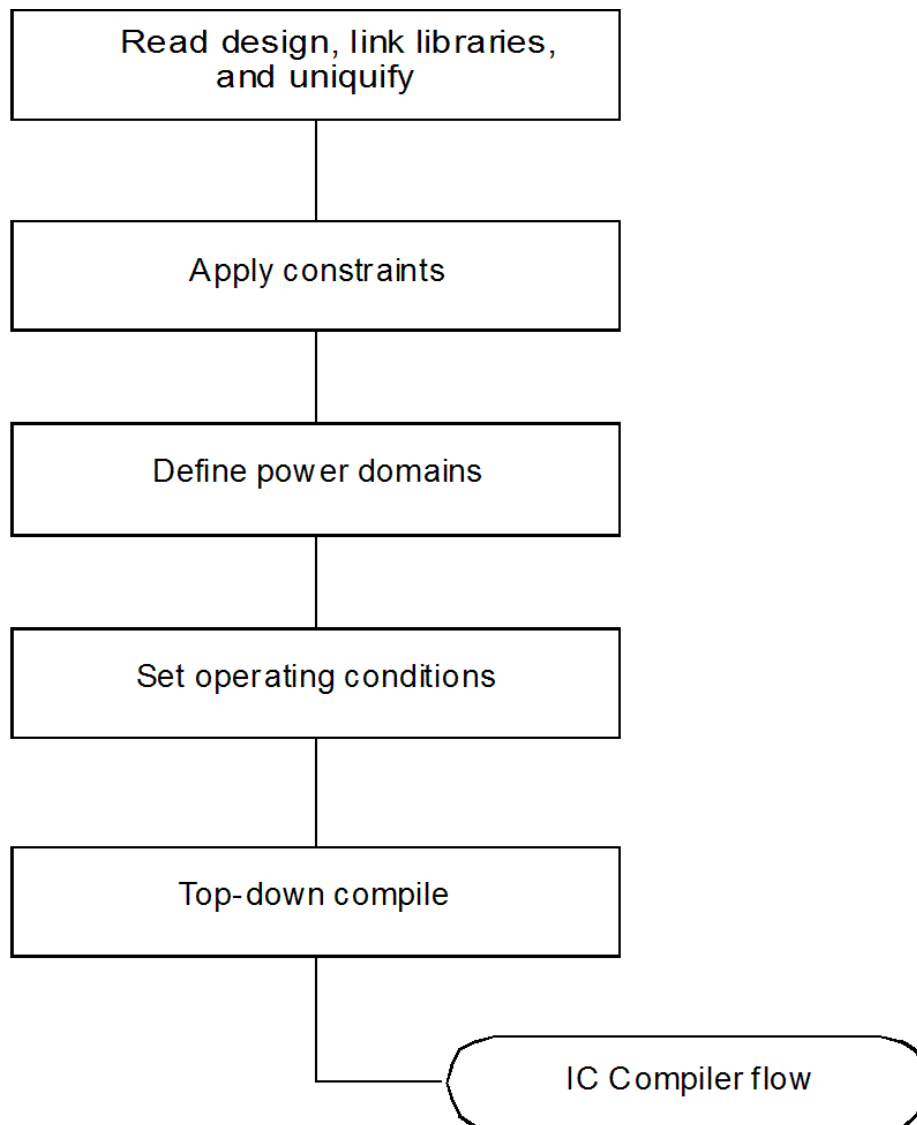
All `dc_shell` commands support multivoltage designs, including:

- `compile_ultra`
- `compile`
- `balance_registers`

- `optimize_registers`
- `pipeline_design`
- `simplify_constants`
- `balance_buffers`
- `clean_buffer_tree`

The basic, high-level synthesis flow is shown in [Figure C-8](#).

Figure C-8 Top-Up Down Flow



Restrictions and Limitations

Note the following restrictions and limitations:

- The `ungroup` command and the `compile_ultra` (or `compile`) command's automatic ungroup do not perform ungrouping on a subblock if the operating condition of the subblock is different from the operating condition of the parent block.
- Retiming is supported only on blocks with uniform operating conditions. This restriction applies to retiming with either the `compile` commands or with the stand alone retiming commands `balance_registers`, `optimize_registers`, and `pipeline_design`.
- The `simplify_constants` command does not simplify across a level shifter because the nets connected to a level shifter are `dont_touch`.
- The `translate` command is not supported for multivoltage designs.
- The `insert_level_shifters` command (if used) requires uniquified designs.
- There is currently no way to specify that two power domains turn on or off simultaneously.
- If power domains are not specified when using isolation cells and enable-type level shifters, an error condition occurs and the run stops.

Handling Designs That Use Isolation Cells and Level Shifters, Including Enable-Type Level Shifters

Additional complexities in the multivoltage design flow arise when the design utilizes both isolation cells and level-shifter cells.

When an isolation cell is instantiated in a design, the tool marks the net that connects the isolation cell to the edge of its power domain as `dont_touch`. If a voltage difference exists on this path and a level shifter cell is needed, the `dont_touch` net can prevent the insertion of the level shifter. However, such isolation cells can be mapped to enable-type level shifter cells, as discussed in the following paragraph.

The design flow is slightly modified to handle this situation. The flow is further altered if enable-type level shifters are available from the technology library, because the tool can map generic technology (GTECH) isolation cells to these enable-type level-shifter cells. The modified flows are outlined in the following sections.

However, before running any of these flows, be sure to:

- Set the `compile_delete_unloaded_sequential_cells` variable to `false`. By default, the `insert_level_shifters` command removes unloaded logic. This variable protects only sequential cells, so if you need to protect other unloaded cells, mark them with the `dont_touch` attribute.

- Run the `uniquify` command after elaboration. The `insert_level_shifters` command requires a uniquified design.

Note:

If you have instantiated both the isolation cells and the level-shifter cells in the RTL design, you do not need to use these modified flows. But you should run the `check_mv_design` command to verify the presence of isolation and level-shifter cells in the RTL design.

Inserting Level Shifters When Enable-Type Level Shifter Library Cells Are Available

If you want to map the GTECH isolation cells to the available enable-type level shifter library cells, you must ensure that

- The design is constrained to require a level shifter on the same net as the isolation path
- Any previously inserted level shifter is not on the same path

The following flows use the `dont_touch` attribute of the isolation cell nets to prevent the `insert_level_shifters` command from inserting buffer-type level shifters on those paths. The `compile_ultra` (or `compile`) command then maps the isolation GTECH cells to an enable-type level shifter.

Dealing With Isolation Cells Inside the Power Domain

Follow these steps to insert buffer-type level shifter cells inside the power domain hierarchy when the isolation cell is inside the power domain,

1. Put a `$isolate` call in the RTL code where you want these cells in the netlist.
2. Use the `create_power_domain` command to create the power domains.
3. Run `set_operating_conditions` on the design.
4. Run `set_operating_conditions` on the power-down hierarchy pins to ensure that the buffer-type level shifters are inserted inside the block.
5. Run `insert_level_shifters` to put buffer-type level shifters on the nonisolated paths inside the power domain.
6. Run any supported compile flow.

The GTECH isolation cells are mapped to the proper enable-type level-shifter cell.

Dealing With Isolation Cells Outside the Power Domain

To insert buffer-type level shifter cells on non-isolated paths outside the power domain when the isolation cells are outside the power domain,

1. Put a `$isolate` call in the RTL code where you want these cells in the netlist.

2. Use the `create_power_domain` command to create the power domains.
3. Run `set_operating_conditions` on the design.
4. Run `insert_level_shifters` to put buffer-type level shifters on the nonisolated paths outside the power domain.
5. Run any supported compile flow.

The GTECH isolation cells are mapped to the proper enable-type level-shifter cell.

Inserting Level Shifters When Enable-Type Level Shifter Library Cells Are *Not* Available

The tool cannot insert level shifters on a net that is `dont_touch` due to the presence of an isolation cell. To fix a voltage violation on such a path, you must constrain the design such that the level shifter is inserted in a different power domain. The following flows show you how to set up the operating conditions to accomplish this.

Dealing With Isolation Cells Outside the Power Domain

To insert buffer-type level shifter cells inside the power domain hierarchy when the isolation cell is outside the power domain,

1. Put a `$isolate` call in the RTL code where you want these cells in the netlist.
2. Use the `create_power_domain` command to create the power domains.
3. Run `set_operating_conditions` on the design.
4. Run `set_operating_conditions` on the ports of the power-down hierarchy pins to ensure that the buffer-type level shifters are inserted inside the block.
5. Run `insert_level_shifters`.
6. Run any supported compile flow.

Dealing With Isolation Cells Inside the Power Domain

To insert buffer-type level shifter cells outside a power domain when the isolation cell is inside the power domain,

1. Put a `$isolate` call in the RTL code where you want these cells in the netlist.
2. Use the `create_power_domain` command to create the power domains.
3. Run `set_operating_conditions` on the design.
4. Run `insert_level_shifters` to put buffer-type level shifters on the nonisolated paths outside the power domain.
5. Run any supported compile flow.

Top-Down Compile Example Script

[Example C-1](#) shows a working top-down compile script for a multivoltage design that uses power domains.

Example C-1 Top-Down Compile Script

```
## script for Design Compiler, version Z-2007.03
## 1) top down compile
## 2) add power domain

#####
##### Set the target library including 0.9/0.7V as well
##### as LS/Isolation library
#####
set target_library
    " abcd90efghijwc.db abcd90efghijwc07.db
      abcd90efghijwc07iso.db abcd90efghijwc0709+.db
      abcd90efghijwc0907+.db "

#####
##### read rtl and insert clock gating
#####
##### global clock gating is used to
#####
##### enable power reduction on CT
#####
##### bypass/ctrl are supposed to operate at
#####
##### 0.9V and will be shut-down after sensing
#####
##### 50 idle clock cycles. dma_fifo_row_top is designed
#####
##### to operate at 0.9V. top level logic will be operating
#####
##### at 0.7V
#####

analyze -format verilog [ glob rtl/*.v]

set_clock_gating_style
    -sequential_cell latch\
    -minimum_bitwidth 2\
    -num_stage 2\
    -positive_edge_logic integrated:CKLNQHVTD1\
    -neg integrated:CKLHQHVTD1\
    -control_point before\
    -control_signal scan_enable

elaborate dma_fifo
```

```

insert_clock_gating -global
report_clock_gating
propagate_constraints -gate_clock

current_design dma_fifo

link

##### Source the SDC constraints #####
#####
source -echo ./sdc/dma_fifo.sdc

#####
##### Creating Power Domains
#####
create_power_domain TOP
create_power_domain BYPASS_PD \
    -object_list bypass_row \
    -power_down \
    -power_down_ctrl sleep_out
create_power_domain CTRL_PD \
    -object_list ctrl \
    -power_down \
    -power_down_ctrl sleep_out
create_power_domain ROW_TOP \
    -object_list dma_fifo_row_top

##### Set The Operating Conditions
#####
##### dma_fifo_row_top @ 0.9V
#####
##### ctrl/bypass_row @ 0.9V Shut-Down
#####
##### top level logic @ 0.7V
#####

set_operating_conditions -max WC07COM\
    -min BCCOM

set_operating_conditions -max WCCOM\
    -min BCCOM\
    -object_list [list bypass_row ctrl]

set_operating_conditions -max WCCOM\
    -min BCCOM\
    -object_list dma_fifo_row_top

set_operating_conditions -max WC07COM\
    -min BCCOM\
    -object_list $bypass_row

set_operating_conditions -max WC07COM\

```

```

                                -min BCCOM\
                                -object_list $dma_fifo_row_top

set_operating_conditions -max WC07COM\
                        -min BCCOM\
                        -object_list $ctrl_outport

set_level_shifter_strategy -rule all
set auto_insert_level_shifter_on_clocks all

check_mv_design -verbose -isolation\
                -level_shifter\
                -opcond_mismatches\
                -target_library_subset\
                -connection_rules >
                ./ $ReportDir/pre_compile.check_mv.rpt
check_mv_design -verbose -level_shifter >
                ./ $ReportDir/pre_compile.check_ls.rpt

#####
##### Top-down compile
#####

compile_ultra -scan
hookup_testports -verbose
insert_dft
check_mv_design -verbose
check_mv_design -verbose -isolation -opcond_mismatches
                -target_library_subset -connection_rules >
                ./ $ReportDir/post_compile.check_mv.rpt

exit

```

Bottom-Up Compile Flow

Note:

The bottom-up compile flow is not a recommended flow although it is supported.

You can carry out the logic synthesis of a design using a bottom-up compile strategy that employs multiple NLDL libraries. That is, you can separately compile the blocks at different voltage levels, with each block referencing the libraries appropriate to that block's voltage level. Within each block, the `dc_allocate_budgets` command is used for RTL-level budgeting. To obtain accurate top-level constraints, budgeting should be performed on unmapped blocks with level shifters inserted.

Note:

For a multivoltage design, if you are compiling a subblock that has only one operating condition, use the `-mv_mode` switch in `dc_shell` to ensure the tool correctly handles multiple NLDM libraries that use same cell names.

The steps for the bottom-up flow are

1. Read the entire design.
2. Link and uniquify the top-level design.
3. Apply top-level constraints, including operating conditions and any target library subsets.
4. Insert buffer-type level shifters to get accurate budgets.
5. Generate block-level RTL budgets by using the `dc_allocate_budgets -mode rtl` command.
6. Compile individual blocks, using the constraints generated in step 4.
7. Compile the top level.

Automated Chip Synthesis Flow

Note:

The Automated Chip Synthesis compile flow is not a recommended flow although it is supported.

You can run Automated Chip Synthesis with multiple NLDM libraries to carry out an automated bottom-up compile of a multivoltage hierarchical design. Buffer-type level shifters are automatically inserted when necessary. For subblocks operating at different voltages, these level shifters are inserted in the netlist before compile. Operating condition rules determine whether level shifters are placed inside or outside the block netlist.

Automated Chip Synthesis uses the default level-shifter strategy and threshold values unless you change these values by using the `set_level_shifter_strategy` and `set_level_shifter_threshold` commands.

The steps for the Automated Chip Synthesis flow are

1. Run the `dc_shell` command.
2. Specify the multiple multivoltage NLDM libraries and the level-shifter and isolation cell libraries in the `link_library` variable.
3. Set operating conditions on the appropriate instances or designs.

4. Run the `uniquify` command if you have different instances of the same block in various designs or instances with different operating conditions.
5. (Optional) Set user-defined partitions by using the `set_compile_partitions` command. You can use the `-level` or `-auto` partitioning option, or you can specify the list of designs to be partitioned.
6. Set the target library for each partition that has its own operating condition and therefore must be mapped to a separate library from its parent by using the `set_target_library_subset` command.
7. Run the `acs_compile_design` command.
8. (Optional) For possible quality-of-results (QoR) improvement, run the `acs_refine_design` command at the top level.

Power Compiler Flows for Multivoltage Designs

Power Compiler power optimization supports multivoltage designs as well as single voltage designs. In particular, this includes the following power optimization flows:

- Clock-gating flow
- Power-gating flow
- Gate-level power optimization flow

You can use the `report_clock_gating` command with the `-gating` option to check the results of these flows.

Clock-Gating Flow

The steps for the clock-gating flow are

1. Set target libraries
2. Set clock-gating style
3. Set variables
4. Read RTL design
5. Read constraints
6. Set operating conditions on blocks
7. Insert clock gates
8. Compile (`compile_ultra` command is recommended)

If clock gating includes test logic, then use the `hookup_testports` command to hookup all the integrated clock-gate test pins to the top-level. This command automatically inserts level shifters on multivoltage designs.

Power-Gating Flow

The steps for the power-gating flow are as follows:

1. Set target libraries
2. Enable power-gating variable
3. Set other variables
4. Read RTL design
5. Read constraints
6. Set operating conditions on blocks
7. Set power-gating style
8. Compile (`compile_ultra` command is recommended)
9. Use power-gating signal specification for correct hookup
10. Hook up power-gating ports
11. Check and insert additional level shifters as needed

In this flow, the `insert_level_shifters` command is required after you run the `hookup_power_gating_ports` command.

Gate-Level Power Optimization Flow

Power Compiler supports all dynamic and leakage optimization flows for multivoltage designs.

The steps for a one-pass leakage optimization flow are as follows:

1. Set target to all multivoltage threshold libraries
2. Read RTL designs
3. Set optimization constraints
4. Set operating conditions on blocks
5. Check preexisting level shifters and remove or preserve as needed
6. Compile (`compile_ultra` command is recommended)

For more information about these power optimization flows, see Chapters 7, 8, and 9.

Multivoltage Elements: \$isolate and \$power

Presto Verilog supports the multivoltage flow by letting you create isolation cells and power domains, using the \$power and \$isolate system tasks. These constructs are described in the following sections:

- [Isolation Cells](#)
 - [Power Domains](#)
-

Isolation Cells

You use isolation cells to isolate the power-down modules from other parts of the design. To create an isolation cell in your RTL, use the \$isolate system task, as shown in [Example C-2](#). Here, the output of the first always block is fed into the isolation cell always block that uses the \$isolate task. This task cannot handle complex port representation.

Example C-2 Using \$isolate

```
module adder(clk, reset, data1, data2, sum, ISO, PwrDwn);
    input      clk, reset;
    input      ISO, PwrDwn;
    input  [7:0] data1, data2;
    output [7:0] sum;

    reg  [7:0] sum_preiso;

    always @(posedge clk or posedge reset)
    begin
        if (reset)
            sum_preiso <= 7'b0;
        else
            sum_preiso <= data1 + data2;
        end

    always @*
    begin
        $isolate(sum, ~ISO, sum_preiso, 1'b1);
    end

endmodule
```

Syntax

```
$isolate(<out>, <enable>, <iso_out>, <data>)
```

<out>: Output of the isolation cell

<enable>: Input to enable the isolation cell

<iso_out>: Input, power-on input of isolation cell

<data>: Input, power-down input of isolation cell

Behavior

The tool builds an isolation cell with the following generic behavior:

```
if (enable)
    out = iso_out;
else
    out = data;
```

Type of Cell Instantiated

The tool instantiates an isolation cell per the following rules:

- When data is constant 0/1, a gtech_isolate cell is instantiated.
- When data is the same as the iso_out, a gtech_isolatch cell is instantiated.

Guidelines

- The tool supports only out/iso_out with a data type of signal (wire) or simple expressions, such as a signal array with a bit index or bit slice index. (The index must be an elaboration time resolvable constant).
- The width of the out and iso_out signals must match.
- The enable signal must be a signal with a scalar type or its negation.
- Data must be an elaboration time constant or the same expression as iso_out. When data is a constant, the width of data can be either 1-bit (the 1-bit value is extended to the width of iso/iso_out) or the same as iso/iso_out.
- The isolate() call cannot exist in an edge-sensitive always block.

Power Domains

Important:

To be synthesized, \$power statements must occur in an initial block, must not be nested within any control structures (such as if, while, and for constructs), and must not be preceded by any timing control statements (#0 for example). This ensures that the \$power construct is always active, matching the simulation behavior.

Example C-3 Using \$power

```
module top(clk, reset, data1, data2, data3, data4, sum, prod);
    input          clk, reset;
    input [7:0]    data1, data2, data3, data4;
    output [7:0]   sum;
    output [15:0]  prod;
```

```

wire    ISO_add, ISO_mult, PwrDwn_add, PwrDwn_mult;
wire    dummy;

reg     dummyq;

adder    add1 (clk, reset, data1, data2, sum, ISO_add, PwrDwn_add);
multiplier mult1 (clk, reset, data3, data4, prod, ISO_mult, PwrDwn_mult);
controller ctrl1 (clk, reset, data1, data2, data3, data4, ISO_add, ISO_mult,
                  PwrDwn_add, PwrDwn_mult);

initial
begin
    $power("PD_adder", PwrDwn_add, 0'b0, "add1");
    $power("PD_mult", PwrDwn_mult, 0'b0, "mult1");
end

endmodule

```

The `$power` construct contains the following information:

Usage Guidelines

Each instance in a design may be named in a `$power` statement exactly once. Once an instance is named in a `$power` statement, that statement completely controls the power behavior of that instance. If an instance is not named in a `$power` statement, its power is controlled by the closest ancestor in the hierarchy explicitly named in a `$power` statement. This restriction ensures that the power hierarchy is explicitly controlled in the `$power` controls, rather than implicitly controlled by having some `$power` statements override others.

For synthesis, it is useful to specify which power domains are always on (power is not removed while the parent is on). To do this with `$power`, set `power_on_net` to a constant, for example:

```
$power("AO_power_domain", 0'b1 0'b1, "pwr_ctrl");
```

Multivoltage Elements: `isolate()` and `power()`

Presto VHDL supports the multivoltage flow by letting you create isolation cells and power domains, using the `isolate()` and `power()` procedures defined in the `snps_ext` package. These procedures are described in the following sections:

- [Isolation Cells](#)
- [Power Domains](#)

To enable these features, you must set `hdl_in_enable_rtl_power_constructs` to `true` (default is `false`) and use the `snps_ext` and `snps_ext.power.all` libraries.

Isolation Cells, Buffers, and Latches

You use isolation procedures to isolate power down designs from other parts of the design. There are two types of `isolate()` procedures: one creates isolation buffers and cells, and the other creates isolation latches. The code in [Example C-4](#) uses the `isolate` procedure to instantiate a GTECH isolate cell.

Example C-4 Using the Isolate Procedure

```
library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.std_logic_unsigned.all;
library snps_ext;
  use snps_ext.power.all;

entity adder is
  port (
    clk : in std_logic;
    reset : in std_logic;
    ISO : in std_logic;
    PwrDwn : in std_logic;
    data1 : in std_logic_vector(7 downto 0);
    data2 : in std_logic_vector(7 downto 0);
    sum : out std_logic_vector(7 downto 0)
  );
end entity adder;

architecture ARCH of adder is
  signal sum_preiso : std_logic_vector(7 downto 0);
begin

  process (clk, reset) is
  begin
    if (reset = '1') then
      sum_preiso <= (others => '0');
    elsif (clk'event and clk = '1') then
      sum_preiso <= data1 + data2;
    end if;
  end process;

  process (ISO, sum_preiso) is
  begin
    isolate(sum, not ISO, sum_preiso, '1');
  end process;
end;
```

Isolation Cell and Buffer Syntax:

```
procedure isolate(<data_out>, <iso>, <data_in>, <clamp>
```

<data_out>: Output of the isolation cell. Data type can be bit, bit_vector, std_logic, std_logic_vector, signed, and unsigned.

<iso>: Input, power-on input of isolation cell. Data type can be bit and std_logic.

<data_in>: Input, power-down input of isolation cell. Data type can be bit, bit_vector, std_logic, std_logic_vector, signed, and unsigned.

<clamp>: Input to enable the isolation cell. Data type can be bit, bit_vector, std_logic, and std_logic_vector.

Coding Style

Use the following coding style to build a GTECH isolation cell or buffer:

```
if (iso = '1') then
    data_out <= data_in;
else
    data_out <= clamp;
end if;
```

When `clamp` is equivalent to 0 or 1, the tool instantiates a GTECH isolate cell; when `clamp` is equivalent to z, the tool instantiates a gtech tri-state buffer.

Isolation Latch Syntax

```
procedure isolate(<data_out>, <iso>, <data_in>
                );
```

<data_out>: Output of the isolation cell. Data type can be bit, bit_vector, std_logic, std_logic_vector, signed, and unsigned.

<iso>: Input, power-on input of isolation cell. Data type can be bit and std_logic.

<data_in>: Input, power-down input of isolation cell. Data type can be bit, bit_vector, std_logic, std_logic_vector, signed, and unsigned.

Coding Style

Use the following coding style to build a GTECH isolation latch:

```
if (iso = '1') then
    data_out <= data_in;
end if;
```

Guidelines

- The tool only supports out/iso_out with a data type of signal (wire) or simple expressions, such as a signal array with a bit index or bit slice index. The index must be an elaboration time resolvable constant.
- The width of the out and iso_out signals must match.
- The enable signal must be a signal with a scalar type or its negation.

- Data must be an elaboration time constant or the same expression as iso_out. When data is a constant, the width of data can be either 1-bit (the 1-bit value is extended to the width of iso/iso_out) or the same as iso/iso_out.
- The isolate() call cannot exist in an edge-sensitive block.

Power Domains

In the multivoltage flow, you use power domains to define the power environment. It includes such items as a power-down condition, the working voltage, or a power net. While much of the power information can be added after RTL compile, some power information needs to be represented in the RTL code so that the verification tool is aware of the power behavior and can therefore simulate the design correctly. To create a power domain in your RTL, use the `power()` procedure, as shown in [Example C-5](#). Here, two power domains, PD_adder and PD_mult, are defined with the `power()` procedure.

Example C-5 Using the Power Procedure

```
library IEEE;
  use IEEE.std_logic_1164.all;
library snps_ext;
  use snps_ext.power.all;
library work;
  use work.components.all;

entity top is
  port (
    clk : in std_logic;
    reset : in std_logic;
    data1 : in std_logic_vector(7 downto 0);
    data2 : in std_logic_vector(7 downto 0);
    data3 : in std_logic_vector(7 downto 0);
    data4 : in std_logic_vector(7 downto 0);
    sum : out std_logic_vector(7 downto 0);
    prod : out std_logic_vector(15 downto 0)
  );
end entity top;

architecture ARCH of top is
  signal ISO_add : std_logic;
  signal ISO_mult : std_logic;
  signal PwrDwn_add : std_logic;
  signal PwrDwn_mult : std_logic;
begin

  add1 : adder
    port map (
      clk => clk,
      reset => reset,
      data1 => data1,
```



```

        data2 => data2,
        sum => sum,
        ISO => ISO_add,
        PwrDwn => PwrDwn_add
    );

    mult1 : multiplier
    port map (
        clk => clk,
        reset => reset,
        data1 => data3,
        data2 => data4,
        prod => prod,
        ISO => ISO_mult,
        PwrDwn => PwrDwn_mult
    );

    ctrl1 : controller
    port map (
        clk => clk,
        reset => reset,
        data1 => data1,
        data2 => data2,
        data3 => data3,
        data4 => data4,
        ISO_add => ISO_add,
        ISO_mult => ISO_mult,
        PwrDwn_add => PwrDwn_add,
        PwrDwn_mult => PwrDwn_mult
    );

    power("PD_adder", PwrDwn_add, '0', "add1");
    power("PD_mult", PwrDwn_mult, '0', "mult1");

end;
```

The `power()` procedure contains the following information:

- Power domain name
- The control signal for power-on and the power-available sense expression
- Power-on and power-available acknowledge expressions
- List of entities in the power domain

Syntax

```

procedure power (<domain_name>, <powerdown_expr>,
<down_sense>,<power_on_net>, <on_sense_expression>,
<power_on_ack_net>, <ack_sense_expression>,
<instance_names_string>);
```

<domain_name>: Name of power domain.

<power_down_expr>: The expression that controls whether the power domain should shut down. When `power_down_expr` is evaluated to the same value as `down_sense`, the power domain shuts down. Otherwise, it stays up.

<down_sense>: The value that determines if the design is taken to a power down state.

<power_on_net>: The control signal for power on. It is limited to simple signals to avoid creation of implicit power domains. A simple signal means that only wiring is required. If the width is greater than 1 bit, the lowest bit is taken. Bus representation for staggered power-on currently is not supported.

<on_sense_expression>: Quoted constant string indicating the power-available sense of the power-on expression. The sense expression must have the same width as the power-on expression in a bit-to-bit correspondence to the power-on expression, and must consist only of 1s and 0s (no X's). Only a single bit string is supported. If the string has more than one bit, only the lowest bit is used.

<power_on_ack_net>: (Optional) The power up acknowledge expression is used to acknowledge (to the outside environment, for example, the power controller) that the power domain is fully up and running. It is limited to simple signals because the procedure drives this net. If the width is greater than one bit, the lowest bit is taken. Bus representation for staggered power-on currently is not supported.

<ack_sense_expression>: (Optional) Quoted constant string indicating the power-available sense of the power acknowledge expression. The sense expression must have the same width as the power acknowledge expression in a bit-to-bit correspondence to the power acknowledge expression, and must only consist of 1s and 0s (no X's). Currently, only single bit strings are supported. If the string has more than one bit, only the lowest bit is used. The sense expressions must be a constant string.

<instance_n>: Only one `<instance_names_string>` argument is allowed. If there are multiple instances in the power domain, they need to be included in a quoted instance name string. Named blocks are not allowed. All the logic contained in the instance is assumed to be part of the defined power domain. Hierarchical module instance reference is not allowed. Quoted name is allowed to create wildcards, and it takes form of "a*", "u*", "...". Exclusion is not supported.

Usage Guidelines

You must use the `snps_ext` and `snps_ext.power.all` libraries.

Each instance in a design may be named in a power procedure exactly once. Once an instance is named, that statement completely controls the power behavior of that instance. If an instance is not named in a power procedure, its power is controlled by the closest

ancestor in the hierarchy explicitly named in a power procedure. This restriction ensures that the power hierarchy is explicitly controlled in the power controls, rather than implicitly controlled by having some power statements override others.

An error is returned if there are any drivers on the `power_on_ack_net` beyond the power procedure statement that defines its value.

To be synthesized, power procedure statements must not be nested within any control structures (such as if, while, and for constructs) and must not be preceded by any timing control statements (#0 for example). This ensures that the power construct is always active, matching the simulation behavior.

Index

Symbols

\$isolate C-49
\$power C-49
\$read_lib_saif 4-12

A

accuracy
 complex cells 6-5
 correlation 6-4
 delay model 6-3
 factors affecting 6-3
acs_compile_design command C-47
acs_refine_design command C-47
add_port_state command 12-15
add_pst_state command 12-16
additional ports 7-69
always-on attribute 12-26
always-on logic
 optimizing C-11
always-on strategy
 setting C-10
analysis
 default switching activity 6-8
analyzing power 6-2
 accuracy 6-2
 annotation, minimum for 5-10
 characterizing the design 6-13
 gate-level 6-5
 power correlation 6-9
 reporting power attributes 6-14
 using report_power 6-6
 with partially annotated designs 6-8
analyzing switching activity 5-10
annotating switching activity
 accuracy effect of 6-2
 analysis 5-10
 creating a clock 5-10
 default switching values 5-12
 estimating unannotated 5-12
 name-mapping database 5-3
 partially versus fully 5-9
 removing annotation 5-11
 with gate-level SAIF 5-4
 with RTL SAIF 5-2
 with set_switching_activity 5-7
annotation
 power analysis minimum requirements 5-10
attributes
 always-on 12-26
 clock gating A-4
 clock_gate_clock_pin A-4
 clock_gate_enable_pin A-4
 clock_gate_out_pin A-4
 clock_gate_test_pin A-4

- clock_gating_integrated_cell A-4
- clock_gate_gicg_basic_cell 7-48
- clock_gating_integrated_cell 7-46
- default_threshold_voltage_group 9-10
- dont use
 - dont_use attribute 7-32
- dont_touch 9-9, C-13, C-16, C-20
- for querying and filtering B-1
- is_clock_gating_cell 7-32
- is_isolation_cell 11-4, C-16
- is_isolation_cell_enable_pin 12-28
- is_level_shifter 11-4, C-16, C-19, C-36
- isolation_cell_enable_pin 11-4
- level_shifter_enable_pin 11-4, C-16
- ok_for_isolation 12-28
- power_gating_pin C-31
- retention_cell 10-4
- retention_pin 10-4
- threshold_voltage_group 9-10
- attributes, library
 - internal_power 3-8
- Automated Chip Synthesis flow C-46
- automatic rollback 8-15

B

- back-annotating, after layout 3-12
- balance_registers command 7-45
- boundary nets
 - constraining in power domains C-11
- buffers
 - clock-tree 6-5
- buffer-type level shifters
 - checking for violations C-19
 - checking the design C-19
 - inserting C-12
 - removing C-20

C

- calculating power 3-4

- capacitive load, obtaining 3-12
- capacitive switching power
 - (see switching power)
- CCS libraries 1-8, 3-16
- CCS power model 1-3
- cell leakage power, equation 3-4
- cells, library
 - complex 6-5
- Channel-Width Model 9-13
- characterize command 6-12
 - Design Compiler 6-12
- characterize design 6-13
- check_design command C-37
- check_isolation_cells command C-37
- check_level_shifters command C-19, C-36
- check_mv_design command 11-7, 12-6, 12-31, C-19, C-35
- check_test command A-4
- clock
 - affect on switching propagation 5-10
- clock constraints, propagating 7-52
- clock domains in clock gating 7-8
- clock gating
 - attributes A-4
 - attributes, querying and filtering B-1
 - choosing cell by name 7-29
 - choosing gating logic 7-24
 - choosing hold time 7-24
 - choosing integrated clock-gating cell 7-30
 - choosing setup time 7-23
 - choosing specific library 7-29
 - clock tree synthesis 7-22
 - conditions for 7-6
 - control point insertion 7-36
 - default style 7-42
 - designating cells 7-32
 - enable condition 7-6, 7-7
 - enhanced register-based 7-70
 - ensuring accuracy, ideal clocks 7-52
 - features 7-3

- gate clock example 7-4
- gated registers 7-44
- generic integrated clock-gating cells 7-31, 7-46
- hierarchical 7-68
- identify_clock_gating usage flow 7-60
- in structural netlists 7-57
- integrated clock-gating cells 7-46
- introduction 7-3
- latched-based 7-51
- latch-free style 7-25, 7-34
- library requirements, gicgs 7-48
- limiting clock gate insertion 7-31
- multistage 7-65
- name conventions, name conventions, clock gating 7-54
- negative edge 7-24
- observability 7-40
- observability depth 7-41
- on DesignWare components 7-72
- overriding clock-gating conditions 7-8
- overriding setup/hold times 7-31
- pin attributes 7-51
- positive edge 7-24
- power-driven clock gating 7-12
- propagating clock constraints 7-52
- remove cell 7-44
- removing clock gates 7-44
- replacing clock-gating cells 7-61
- rewiring after retiming 7-45
- sample script 7-52
- scan enable versus test mode 7-38
- script, naming style 7-56
- script, output netlist 7-56
- selecting clock-gating style 7-33
- selecting latches 7-34
- setup and hold times, specifying 7-19
- setup condition 7-6, 7-7
- target library 7-29
- test port naming 7-38
- testability, improving 7-36
- timing analysis 7-14
- timing considerations 7-51
- ungrouped clock gates 7-57
- width condition 7-7
- with operand isolation 8-19
- write_script usage flow 7-59
- clock gating cell
 - deleting 7-44
 - specifying 7-44
- clock gating registers
 - moving 7-44
- clock tree synthesis, impact on setup and hold times 7-22
- clock_gate_clock_pin attribute A-4
- clock_gate_enable_pin attribute A-4
- clock_gate_gicg_basic_cell attribute 7-48
- clock_gate_out_pin attribute A-4
- clock_gate_test_pin attribute A-4
- clock_gating_integrate_cell attribute 7-46
- clock_gating_integrated_cell attribute A-4
- clock_gating_integrated_cell command 7-46
- clock-gating
 - latch-free 7-52
- clock-gating multivoltage flow 11-7, C-47
- clock-tree buffers 6-5
- command
 - reading a design 1-7
 - syntax 1-8
 - ways to enter 1-8
 - writing a design 1-7
- command-line interface
 - introduction to 1-6
 - quitting 1-6
 - starting 1-6
- commands
 - acs_compile_design C-47
 - acs_refine_design C-47
 - add_port_state 12-15
 - add_pst_state 12-16
 - balance_registers 7-45
 - change_names
 - change_names command 7-59

characterize 6-12, 6-13
check_design C-37
check_isolation_cells C-37
check_level_shifters C-19, C-36
check_mv_design 11-7, 12-6, 12-31, C-19, C-35
check_test A-4
clock_gating_integrated_cell 7-46
connect_power_domain C-6
connect_power_net_info C-4, C-7
connect_supply_net 12-13
create_operating_condition C-35
create_power_domain 11-3, 12-10, C-4, C-5
create_power_net_info C-4, C-6
create_power_switch 12-14
create_pst 12-16
create_supply_net 12-13
create_supply_port 12-12, 12-13
create_voltage_area 12-28
current_instance C-27
dc_allocate_budgets C-45
disconnect_power_net_info C-7
for operand isolation 8-11
get_always_on_logic C-8
get_attribute 9-5
get_power_domains C-5
hookup_power_gating_ports C-21
identify_clock_gating 7-61
infer_power_domain 11-3, C-4
insert_clock_gating 7-9, 7-42
insert_dft 7-42
insert_level_shifters C-12, C-13
map_isolation_cell 12-22
map_level_shifter_cell 12-20
map_retention_cell 12-24
merge_saif 5-4, 5-6
optimize_registers 7-45
power_gating_pin C-28
power_model_preference 1-8
read_saif 5-2, 5-4, 5-5
remove_attributes 9-5
remove_clock_gating 7-44
remove_level_shifters C-20
remove_operand_isolation 8-16
remove_power_domain C-5
remove_power_net_info C-6
remove_target_library_subset 11-7, 12-6
replace_clock_gates 7-61
report_cell C-37
report_constraint 14-24
report_delay_calculation C-37
report_hierarchy C-38
report_isolation_cell 12-30
report_level_shifter 12-29
report_lib 6-12, 6-14
report_operand_isolation 8-18
report_operating_conditions C-35
report_power 5-2, 6-2, 6-6, 6-15
report_power_domain 12-29, C-5
report_power_gating C-32
report_power_net_info C-6
report_power_pin_info C-5
report_power_pin_information C-7
report_power_switch 12-30
report_pst 12-30
report_qor 14-22
report_retention_cell 12-31
report_saif 5-10
report_supply_net 12-31
report_supply_port 12-31
report_target_library_subset 11-7, 12-6, 12-31, C-35
report_threshold_voltage_group 9-11
report_timing C-37
report_timing -scenario 14-23
report_tlu_plus_files 14-25
reset_switching_activity 5-11
rewire_clock_gating 7-44, 7-45
saif_map 5-2
set_always_on_strategy C-10
set_attribute 9-10, 11-5, 11-6, 12-25, 12-28, C-8, C-10
set_cell_internal_power 6-5
set_clock_gating_registers 7-8

- set_clock_gating_style 7-6, 7-24, 7-30, 7-33
- set_clock_transition 7-52
- set_compile_partitions C-47
- set_domain_supply_net 12-14
- set_dont_use 7-31
- \$set_gate_level_monitoring 4-12
- set_isolation 12-20
- set_isolation_control 12-22, 12-27
- set_leakage_power_model 9-12
- set_level_shifter 12-18
- set_level_shifter_strategy C-46
- set_level_shifter_threshold C-19, C-46
- set_max_dynamic_power 6-10, 9-5, 9-15
- set_max_leakage_power 6-10, 9-5, 9-9, 14-5
- set_max_total_power 6-10
- set_min_library 14-11
- set_operand_isolation_cell 8-14
- set_operand_isolation_scope 8-12
- set_operand_isolation_slack 8-15
- set_operand_isolation_style 8-12
- set_operating_conditions 14-3, 14-8, 14-9, C-14
- set_power_gating_signal C-20, C-28
- set_power_gating_style 6-10, C-20
- set_power_prediction 6-10
- set_relative_always_on C-3, C-7
- set_retention 12-23
- set_retention_control 12-24, 12-27
- set_scope 12-10
- set_switching_activity 5-7, 5-9, 5-11
- set_target_library_subset 11-4, 11-6, 12-6, C-47
- set_tlu_plus_files 14-5
- set_voltage C-9
- toggle control examples 4-19
- \$toggle_report 4-10
- \$toggle_reset 4-11
- uniquify C-47
- write_script 6-12, 6-15, 7-59
- compile_delete_unloaded_sequential_cells variable C-40

- complex cells
 - accuracy 6-5
- composite current source power model 1-3
- connect_power_domain command C-6
- connect_power_net_info command C-4, C-7
- connect_supply_net command 12-13
- connect_supply_port command 12-13
- connecting test ports 7-41
 - using hookup_test_ports 7-41
 - using hookup_testports 7-41
- control points, for testability 7-36
- correlation
 - defined 6-4
- cps_default_sp 6-8
- cps_default_tr 6-9
- create_operating_condition command C-35
- create_power_domain command 11-3, 12-10, C-4, C-5
- create_power_net_info command C-4, C-6
- create_power_switch command 12-14
- create_pst command 12-16
- create_supply_net command
 - UPF commands
 - create_supply_net 12-13
- create_supply_port command 12-12
- create_voltage_area command 12-28
- current_instance command C-27

D

- data flow
 - figure 2-4
 - overview 2-3
- datapath designs 8-2
- dc_allocate_budgets command C-45
- dc_shell interface 1-6, 1-7
- dc_shell-topo interface 1-7, 6-9
- debugging, operand isolation 8-21
- default switching activity values 5-12

- default type, power gating C-22
- default value 6-8
- default_threshold_voltage_group attribute 9-10
- defaults
 - switching activity 6-8
- definitions
 - C_{Loadi} 3-12
 - correlation 6-4
 - dynamic power 3-2
 - integrated clock-gating cell 7-46
 - internal power 3-3
 - path-dependent power 3-11
 - power correlation 6-9
 - power gating C-20
 - retention registers 10-2
 - short circuit power 3-3
 - static power 3-2
 - static probability 4-2
 - switching activity 4-2
 - switching power 3-2
 - toggle rate 4-2
- delay model
 - accuracy
 - effect on 6-3, 6-4
 - glitching 6-4
- derived attributes B-2
- Design Compiler commands
 - characterize 6-12
- design cycle, with power 2-2
- design exploration
 - Power Compiler 6-11
- design flow 2-1
- design rule cost function 9-6
- design rule fixing 9-8
- designs
 - reading in 1-7
 - writing out 1-7
- DesignWare components, clock gating 7-72
- disconnect_power_net_info command C-7
- don't care conditions 8-3

- dont_touch attribute 9-9, C-13, C-16, C-20
- dp format
 - creating with toggle_report 4-10
- dp_shell
 - interface 1-6, 1-7
- dynamic power 3-2
 - defined 3-2
 - library requirements 1-6
- dynamic power optimization 9-14
 - performing 9-15
 - switching activity annotation 9-15
- dynamic power optimization
 - scripts, sample 9-15

E

- enable condition, clock gating 7-6, 7-7
- enable power optimization 2-5
- enhanced register-based clock gating 7-70
- entering commands
 - methods of 1-8
 - syntax format 1-8
- equations
 - internal power 3-9
 - leakage power 3-4
 - switching power 3-12
- exiting the command-line interface 1-6
- extensions, defaults for output files 1-8

F

- features
 - clock gating 7-3
 - Power Compiler 1-5
- format, input
 - list of 1-8
 - requiring license keys 1-8

G

- gated clocks, and timing analysis 7-14

- gate-level multivoltage optimization 11-8, C-48
- gate-level power analysis 6-5
- gate-level power optimization 9-1, 9-2
 - attributes 9-5
 - commands for 9-5
 - constraints 9-5
 - cost priority 9-6
 - design rule constraints 9-6
 - design rule fixing 9-8
 - dynamic 9-15
 - flow 9-2
 - in synthesis flow 9-4
 - incremental optimization
 - incremental optimization 9-8
 - leakage power optimization 9-9
 - optimization constraints 9-6
 - performing 9-5
 - positive timing slack 9-7
 - power constraints 9-6
 - unmet constraints 9-8
- gate-level simulation
 - Power Compiler interface
 - Verilog 4-7
 - state- and path dependent switching activity 4-12
- generating SAIF files 4-1
- generic integrated clock-gating cells 7-31, 7-46
 - library-level requirements 7-48
- get_always_on_logic command C-8
- get_attribute command 9-5
- get_power_domains command C-5
- glitching
 - zero-delay model 6-4

H

- hdlin_enable_rtl_power_constructs variable C-51
- help
 - command, for a 1-9
 - displaying for a topic 1-10

- displaying man pages 1-9
- hierarchical clock gating 7-68
- hierarchical levels
 - report 6-7
- hold time 7-31
- hold time, choosing 7-24
- hold time, specifying 7-19
- hookup_power_gating_ports command C-21

I

- ideal clocks 7-52
- identify_clock_gating command 7-61
- identifying power and accuracy 6-2
- infer_power_domain command 11-3, C-4
- input formats 1-8
- inputs
 - primary
 - default switching for 6-8
- insert_clock_gating command 7-9, 7-42
- insert_dft command 7-42
- insert_level_shifters command C-12, C-13
 - uniquified designs required 11-7, C-5, C-40
- instance-based mapping, power gating C-27
- integrated clock-gating cell
 - choosing the cell 7-30
 - example A-1
 - generic cell 7-31, 7-46
- integrated clock-gating cells 7-46
- interface, command line 1-6
- internal nets, monitoring 4-12
- internal power 1-6, 3-2
 - attribute 3-8
 - calculation 3-8
 - defined 3-3
 - equations for summing 3-9
 - look-up table, figure 3-10
 - look-up tables 3-10
 - modeling 3-3, 3-9
 - path-dependent, defined 3-11

- rise and fall, separate 3-12
- short circuit power 3-3
- interoperability, operand isolation 8-19
- introduction 9-2
- Introduction to UPF 12-1
- is_clock_gating_cell attribute 7-32
- is_isolation_cell attribute 11-4, C-16
- is_isolation_cell_enable_pin attribute 12-28
- is_level_shifter attribute 11-4, C-16, C-19, C-36
- I_{sc} , short circuit current 3-3
- isolate() C-51, C-53
- isolation cell
 - definition 11-3
 - GTECH isolation cells C-17
 - inserting C-16
- isolation cells C-49
- isolation_cell_enable_pin attribute 11-4

K

- k-factors
 - multivoltage designs 11-4, 12-5
 - unsupported in multicorner-multimode 14-10

L

- latch-based clock gating 7-51
- latch-free clock gating 7-8, 7-34, 7-52
- latch-free clock-gating style 7-25
- leakage power 1-6
 - calculation 3-4
 - cause of 3-2
 - equation 3-4
 - library requirements 1-6
 - modeling 3-2
- leakage power calculation
 - channel width model 9-12
 - choosing a model 9-12

- default model 9-12
- leakage power optimization 9-9
 - power critical range 9-14
 - sample scripts 9-13
- leakage power optimization, with multivoltage libraries 9-10
- level shifter
 - definition 11-3
 - inserting enable-type C-16
- level_shifter_enable_pin attribute 11-4, C-16
- libraries
 - attributes
 - internal_power 3-8
 - CCS 1-3, 1-8, 3-16
 - NLPM 1-8
 - reporting power attributes of 6-14
 - requirements for optimization 1-6
- library
 - complex cells 6-5
- library models 1-3
- library preparation
 - shut-down blocks 11-4
- library requirements
 - multivoltage designs 11-4, 12-5
- license requirements 1-7
- licenses
 - input formats requiring 1-8
- link_library variable C-46
- look-up tables 3-10 to 3-11

M

- macro cells, as power domains C-8
- man pages, displaying 1-9
- manual rollback 8-16
- map_isolation_cell command 12-22
- map_level_shifter_cell command 12-20
- map_retention_cell command 12-24
- mapping retention registers C-22
- merge_saif command 5-4, 5-6

- methodology flows, figures
 - Verilog gate-level simulation 4-8
 - Verilog RTL simulation 4-7
 - Verilog toggle commands 4-9
- methodology, gate-level
 - state- and path dependent switching 4-12
 - Verilog simulation 4-9
- methodology, RTL
 - Verilog simulation 4-9
 - VSS simulation A-1, B-1
- minimum_bitwidth 7-7
- modeling delay 6-3, 6-4
- modeling power
 - internal power 3-9
 - introduction to 1-3
 - leakage power 3-2
 - lookup table, three-dimensional 3-10
 - switching power 3-12
- modelling retention register cells 10-4
- monitoring internal nets 4-12
- multicorner-multimode 14-2
 - Basic Flow 14-3
 - concepts 14-2
 - handling libraries 14-6
 - k-factors, unsupported 14-10
 - Optimization 14-2
 - optimizing for dynamic power 14-18
 - optimizing for leakage power 14-16
 - report_constraint command 14-24
 - report_qor command 14-22
 - report_timing -scenario command 14-23
 - report_tlu_plus_files command 14-25
 - reporting commands 14-19
 - reporting examples 14-22
 - scenario definition 14-2
 - scenario management 14-15
 - script example 14-29
 - set up 14-5
 - set_max_dynamic_power command 14-16
 - set_max_leakage_power command 14-5
 - set_min_library command 14-11
 - set_tlu_plus_files command 14-5
 - setup considerations 14-5
 - supported SDC commands 14-28
- multistage clock gating 7-65
- Multivoltage
 - UPF flow for multivoltage design implementation 12-1
- multivoltage and multisupply designs 11-2
 - definitions 11-2
 - power domains 11-2
- multivoltage compile flow
 - Automated Chip Synthesis, top-down C-46
 - Design Compiler, bottom-up C-45
 - Design Compiler, top-down C-38
 - limitations C-40
 - script C-43
 - using isolation cells and level shifters C-40
- multivoltage designs
 - Hierarchical UPF Flow 12-32
 - k-factors 11-4, 12-5
 - library requirements 11-4, 12-5
 - PG pin 12-5
 - power and ground pin syntax 12-5
 - shut-down blocks
 - library preparation 11-4
 - target library subsetting 12-6
- multivoltage domain
 - clock-gating flow 13-2
 - gate-level power optimization flow 13-3
 - isolation constructs C-1
 - power constructs C-1
 - power optimization 13-1
 - reporting 13-4
- multivoltage libraries 9-10
- multivoltage power compiler flow
 - clock-gating 11-7, C-47
 - gate-level optimization flow 11-8, C-48
 - power-gating 11-8, C-48

N

- name blocks, power gating C-24
- name-mapping database 5-3
- name-mapping, and PrimeTime PX 5-4
- negative edge 7-24
- net switching power equation 3-12
- NLDM models 3-9
- NLPM libraries 1-8
- nonlinear delay model libraries
 - requirements 11-4
 - using C-38, C-45, C-46

O

- observability depth, choosing 7-41
- observability don't care conditions 8-3
- observability, clock gating
 - circuitry, figure 7-40
 - logic depth, choosing 7-39
- observability, increasing 7-39
- ok_for_isolation attribute 12-28
- one-pass operand isolation 8-7
- online
 - help 1-9
 - man pages 1-9
- operand isolation
 - attributes, querying and filtering B-1
 - automatic rollback 8-15
 - commands for 8-11
 - conditions for 8-3
 - debugging 8-21
 - examples 8-22
 - in Power Compiler 8-3
 - insertion mode 8-12
 - interoperability 8-19
 - introduction 8-2
 - manual rollback 8-16
 - methodology flows 8-4
 - observability don't care conditions 8-3
 - one-pass flow 8-7

- reporting 8-18
- rollback 8-15
- scope 8-12
- scripts, sample 8-9, 8-16
- style 8-12
- two-pass flow 8-5
- user directives 8-14
- variables for 8-12
 - with clock gating 8-19
 - with testability 8-20
- operand isolations
 - reporting 8-23
- optimization, power
 - minimum annotation requirements 5-10
- optimize registers command 7-45
- options, command
 - clock 5-7
 - period 5-7
- output files
 - default extensions 1-8
 - SAIF format example 4-20

P

- path-dependent power, defined 3-11
- path-dependent switching activity
 - library requirements 1-6
 - path-dependent power 3-11
- performing power analysis 6-1
- phsyopt_power_critical_range variable 9-14
- Physical Compiler
 - clock gating cell
 - specifying 7-44
- pin attributes 7-51
- pins, wiring C-28
- positive edge 7-24
- positive timing slack 9-7
- power
 - calculations 3-4
 - internal cell 3-8
 - leakage, modeling 3-2

- switching power, defined 3-12
- power analysis
 - associated clock 6-8
 - characterizing the design 6-13
 - creating a clock 6-6
 - gate level 6-5
 - impact of annotation on 6-2
 - introduction 6-1
 - invoking 6-5
 - minimum annotation requirements 5-10
 - power correlation 6-9
 - report power attributes 6-14
 - report_power command 6-6
 - switching activity defaults 6-8
 - using report_power 6-6
 - with partially annotated designs 6-8
- power analysis and reporting 2-5
- power analysis technology 1-3
- Power Compiler
 - design exploration 6-11
 - design flow 2-1
 - feature overview 1-5
 - features of 1-5
 - gate-level netlist formats 1-8
 - in power methodology 2-4
 - in power methodology, figure 2-4
 - interdependencies of inputs 6-4
 - interface to Verilog, methodology 4-8
 - introduction 1-1
 - license requirements 1-7
 - methodology 1-2
 - operand isolation 8-3
 - optimization and analysis flow 2-3
 - power analysis features 1-4
 - required inputs for 1-5
 - UPF flow for multivoltage design
 - implementation 12-1
 - user interface, introduction to 1-6
- power constraints, gate-level power optimization 9-6
- power correlation 6-10
- power correlation, defined 6-9
- power correlation, performing 6-10
- power critical range 9-14
- power dissipation, figure 3-4
- power domains
 - connecting C-6
 - constraining boundary nets C-11
 - creating C-4
 - creating power net information C-6
 - defining relative always-on C-3, C-7
 - definition 11-2
 - finding always-on paths C-8
 - inferring from the RTL design C-4
 - macro cells C-8
 - power down domains C-3
 - preparing multivoltage libraries
 - marking always-on leaf cell instances 11-6
 - marking always-on library cells 11-5
 - marking pass-gate library pins 11-6
 - relative always-on C-7
 - reporting power pin information C-7
 - specifying exceptional power net
 - connections C-7
 - voltage areas 11-3
- power gating
 - default type C-22
 - defined C-20
 - design-level mapping C-24
 - instance-based mapping C-27
 - mapping retention registers C-22
 - MTCMOS retention registers C-20
 - multithreshold-CMOS retention registers 10-1
 - named blocks C-24
 - pin writing rules C-28
 - wiring pins C-28
- power methodology
 - enable power optimization 2-5
 - overview 2-3
 - power analysis and reporting 2-5
 - simulation 2-5
 - synthesis and power optimization 2-5

- power modeling
 - dynamic power 3-2
 - features supported 1-3
 - internal power 3-8
 - static power 3-2
 - switching power 3-12
- power optimization
 - attributes, querying, filtering B-1
 - design exploration 6-11
 - dynamic 9-14
 - flow 2-4
 - gate-level 9-1
 - minimum annotation requirements 5-10
 - multivoltage domain 13-1
- power rails
 - specifying voltages C-9
- power types, defined 3-2
- power() C-51, C-55
- power_cg_cell_naming_style variable 7-55
- power_cg_flatten variable 7-57
- power_cg_gated_clock_naming_style variable 7-55
- power_cg_module_naming_style variable 7-55
- power_cg_print_enable_conditions variable 7-7
- power_cg_print_enable_conditions_max_terms variable 7-7
- power_default_static_probability variable 5-13, 6-8
- power_default_toggle_rate variable 5-13, 6-9
- power_default_toggle_rate_type variable 6-9
- power_enable_one_pass_power_gating variable C-21
- power_gating_pin attribute C-31
- power_gating_pin command C-28
- power_model_preference command 1-8
- power_model_preference variable 3-16
- power-driven clock gating 7-12
- power-gating multivoltage flow 11-8, C-48
- PrimeTime PX, integration with 5-4

- propagating clock constraints 7-52
- propagating switching activity 5-13
- P_{sc}, short circuit power 3-3

Q

- quitting the command-line interface 1-6

R

- read command 1-8
 - (*see also* reading designs)
- Read RTL Design 7-70
- \$read_lib_saif 4-12
- read_saif command 5-2, 5-4, 5-5
- reading designs
 - read command 1-7
- recompiling
 - after characterizing 6-14
- registers
 - gated
 - moving 7-44
- relative always-on power domains C-3
- remove
 - clock gating cell 7-44
- remove_attributes command 9-5
- remove_clock_gating command 7-44
- remove_level_shifters command C-20
- remove_operand_isolation command 8-16
- remove_power_domain command C-5
- remove_power_net_info command C-6
- remove_target_library_subset command 11-7, 12-6
- removing switching activity annotation 5-11
- replace_clock_gates command 7-61
- replacing clock-gating cells 7-61
- report_cell command C-37
- report_command 6-6
- report_constraint command 14-24
- report_delay_calculation command C-37

- report_hierarchy command C-38
 - report_isolation_cell command 12-30, 12-31
 - report_level_shifter command 12-29
 - report_lib command 6-12, 6-14
 - report_operand_isolation command 8-18
 - report_operating_conditions command C-35
 - report_power command 5-2, 6-2, 6-15
 - syntax 6-6
 - report_power_domain command 12-29, C-5
 - report_power_gating command C-32
 - report_power_net_info command C-6
 - report_power_pin_info command C-5
 - report_power_pin_information command C-7
 - report_power_switch command 12-30
 - report_pst command 12-30
 - report_qor command 14-22
 - report_saif command 5-10
 - report_supply_net command 12-31
 - report_supply_port command 12-31
 - report_target_library_subset command 11-7, 12-6, 12-31, C-35
 - report_threshold_voltage_group command 9-11
 - report_timing command C-37
 - report_timing -scenario command 14-23
 - report_tlu_plus_files command 14-25
 - reporting library attributes 6-14
 - reporting toggle count
 - Verilog simulation 4-10
 - reports
 - analysis effort high 6-15
 - cell 6-18
 - clock gating, multivoltage domain 13-4
 - cumulative 6-18
 - flat 6-17
 - hierarchical 6-19
 - hierarchical levels 6-7
 - net 6-17
 - nworst 6-17
 - operand isolation 8-18, 8-23
 - retention registers C-32
 - sort mode
 - net switching power 6-17
 - sort mode, cumulative fanout 6-18
 - threshold voltage 9-11
 - verbose 6-15
 - reset_switching_activity command 5-11
 - retention register
 - cell attributes 10-4
 - interoperability 10-5
 - modelling cells in library 10-4
 - pin attributes 10-4
 - retention registers 10-4
 - description 10-2
 - design-level mapping C-24
 - mapping C-22
 - MTCMOS C-20
 - multithreshold-CMOS 10-1
 - pin wiring rules C-28
 - purpose of 10-2
 - registers not to map C-27
 - remapping C-27
 - reporting C-32
 - script, flow C-21
 - using 10-3
 - wiring use models C-30
 - retention_cell attribute 10-4
 - retention_pin attribute 10-4
 - rewire_clock_gating command 7-44, 7-45
 - rise and fall power 3-12
 - rollback, operand isolation 8-15
 - RTL isolation constructs
 - multivoltage domain
 - isolation constructs C-1
 - RTL power constructs C-1
 - rules, pin wiring C-28
- ## S
- SAIF
 - annotating switching activity 5-1

- annotation, integration with PrimeTime PX
 - 5-4
- defined 4-2
- generating 4-1
- generating with VCD 4-4
- generation, RTL Verilog methodology 4-6
- generation, Verilog gate-level methodology 4-7
- output file, example 4-20
- reading 5-4, 5-6
- Verilog simulation
 - writing SAIF files 4-10
- saif_map command 5-2
- scan enable 7-38
- scan_enable 7-42
- scenario definition
 - multicorner-multimode 14-2
- script, sample, clock gating 7-52
- scripts
 - using for power analysis 6-15
 - write_script command 6-13, 6-15
- SDPD
 - (see state- and path-dependent)
- separate rise and fall power, specifying 3-12
- set_always_on_strategy command C-10
- set_attribute command 9-10, 11-5, 11-6, 12-25, 12-28, C-8, C-10
- set_cell_internal_power command 6-5
- set_clock_gating_registers command 7-8
- set_clock_gating_style command 7-6, 7-24, 7-30, 7-33
- set_clock_transition command 7-52
- set_compile_partitions command C-47
- set_cope command 12-10
- set_domain_supply_net command 12-14
- set_dont_use command 7-31
- \$set_gate_level_monitoring 4-12
- set_isolation command 12-20
- set_isolation_control command 12-22, 12-27
- set_leakage_power_model command 9-12
- set_level_shifter command 12-18
- set_level_shifter_strategy command C-46
- set_level_shifter_threshold command C-19, C-46
- set_max_dynamic_power command 6-10, 9-5, 9-15
- set_max_leakage_power command 6-10, 9-5, 9-9
- set_max_total_power command 6-10
- set_min_library command 14-11
- set_operand_isolation_cell 8-14
- set_operand_isolation_scope command 8-12
- set_operand_isolation_slack command 8-15
- set_operand_isolation_style 8-12
- set_operating_conditions command C-14
- set_operating_conditions command 14-3, 14-8, 14-9
- set_power_gating_signal command C-20, C-28
- set_power_gating_style command 6-10, C-20
- set_power_prediction command 6-10
- set_relative_always_on command C-3, C-7
- set_retention command 12-23
- set_retention_control command 12-24, 12-27
- set_switching_activity command 5-7, 5-9, 5-11
- set_target_library_subset command 11-4, 11-6, 12-6, C-47
- set_tlu_plus_files command 14-5
- \$set_toggle_region 4-13
- set_voltage command C-9
- setup condition, clock gating 7-7
 - clock gating conditions, list 7-6
- setup time 7-31
- setup time, choosing 7-23
- setup time, specifying 7-19
- short circuit current, I_{sc} 3-3
- short circuit power
 - defined 3-3
 - internal power 3-3

- symbol for 3-3
- shut-down blocks
 - library preparation 11-4
- simulation
 - in power methodology 2-5
 - internal zero-delay 6-6
- simulation, gate-level
 - path-dependent switching activity 4-12
 - state-dependent switching activity 4-12
- starting the command-line interface 1-6
- state and path dependency 3-11
- state- and path-dependent switching activity
 - deriving 5-13
 - library requirements for 1-6
- state- and path-dependent toggle rates 5-2
- state-dependent static probability 5-2
- state-dependent switching activity 1-6
- state-dependent toggle rates 5-2
- static power 3-2
- static probability 6-8
 - default
 - changing 6-8
 - defined 4-2
- switching activity
 - analyzing 5-10
 - annotateable 5-2
 - annotating 5-1
 - annotating partially versus fully 5-9
 - annotating with default values 5-12
 - annotating with gate-level SAIF 5-4
 - annotating with RTL SAIF 5-2
 - annotating with set_switching_activity 5-7
 - creating a clock 5-10
 - defaults 6-8
 - defined 4-2
 - estimating unannotated 5-12
 - for dynamic power optimization 9-15
 - gate-level simulation
 - state- and path-dependent 4-12
 - importance of 5-2
 - name-mapping database 5-3

- propagating 5-13
- removing 5-11
- Switching Activity Information Format (SAIF)
 - files 4-1
- switching activity, annotating
 - accuracy 6-2
- switching activity, capturing
 - Verilog simulation
 - gate-level 4-7
- switching power 3-2
 - equation for 3-12
- switching power, defined 3-2
- synchronous load-enable
 - in a register bank 7-3
- syntax
 - general 1-8
- synthesis and power optimization 2-5

T

- target library subsets
 - using 11-6
- target library subsetting, multivoltage designs 12-6
- target_library variable 11-7, 12-6
- test mode 7-38
- test port naming 7-38
- test_mode 7-39, 7-42
- testability, clock gating 7-36
- testability, inserting control points 7-36
- testability, with operand isolation 8-20
- testbench
 - Verilog example 4-19
 - Verilog, toggle command flow 4-9
- threshold voltage 9-11
- threshold_voltage_group attribute 9-10
- timing analysis, with gated clocks 7-14
- timing considerations, clock gating 7-51
- toggle commands
 - examples 4-19

- toggle rate
 - default
 - changing 6-8
 - default value 6-8
 - defined 4-2
 - toggle_rate and TR, difference 5-7
- \$toggle_report 4-10
- \$toggle_report command 4-10
- \$toggle_reset command 4-11
- topographical domain (in Design Compiler)
 - 1-7, 6-9
- two-pass operand isolation 8-5

U

- unannotated switching activity, estimating 5-12
- Unified Power Format 12-1
- uniquified designs, insert_level_shifters
 - command 11-7, C-5, C-40
- uniquify command C-47
- unmet constraints, gate-level power
 - optimization 9-8
- UPF
 - adding port state information to supply ports
 - 12-15
 - basic AND and OR gates as isolation cells
 - 12-27
 - connecting supply nets 12-14
 - creating power domains 12-10
 - creating power state table 12-16
 - creating power switch 12-14
 - creating supply nets 12-13
 - creating supply port 12-12
 - defining multivoltage design strategies 12-17
 - defining power domains and supply network
 - 12-10
 - handling always-on logic 12-25
 - Hierarchical Flow 12-32
 - hierarchy and scope 12-10
 - mapping retention registers 12-24
 - multivoltage design flow using UPF 12-7
 - multivoltage design implementation 12-1

- specifying primary supply for power domains
 - 12-14

UPF commands

- add_port_state 12-15
- add_pst_state 12-16
- create_power_domain 12-10
- create_power_switch 12-14
- create_pst 12-16
- create_supply_port 12-12
- map_retention_cell 12-24
- set_domain_supply_net 12-14
- set_isolation 12-20
- set_isolation_control 12-22
- set_level_shifter 12-18
- set_retention 12-23
- set_retention_control 12-24
- set_scope 12-10

UPF concepts

- always-on logic cells 12-4
- isolation cells 12-3
- level-shifter 12-3
- PG pin 12-5
- power and ground pin syntax 12-5
- power domains 12-2

UPF strategy

- isolation 12-20
- level shifter 12-18
- retention 12-23

- user directives, operand isolation 8-14

user interface

- Power Compiler, introduction to 1-6

V

variables

- compile_delete_unloaded_sequential_cells
 - C-40
- cps_default_sp 6-8
- cps_default_tr 6-9
- for operand isolation 8-12
- hdlin_enable_rtl_power_constructs C-51
- link_library C-46

- physopt_power_critical_range 9-14
 - power_cg_cell_naming_style 7-55
 - power_cg_flatten 7-57
 - power_cg_gated_clock_net_naming_style 7-55
 - power_cg_module_naming_style 7-55
 - power_cg_print_enable_conditions 7-7
 - power_cg_print_enable_conditions_max_terms 7-7
 - power_default_static_probability 5-13, 6-8
 - power_default_toggle_rate 5-13, 6-9
 - power_default_toggle_rate_type 6-9
 - power_enable_one_pass_power_gating C-21
 - power_model_preference 3-16
 - target_library 11-7, 12-6
 - Verilog language
 - testbench 4-19
 - Verilog netlist 7-54
 - Verilog simulation
 - capturing switching activity
 - gate-level 4-7
 - example files, gate-level 4-18
 - example files, RTL 4-14
 - specific time periods, defining 4-11
 - Verilog testbench 4-19
 - voltage areas
 - power domains 11-3
- ## W
- width condition, clock gating 7-7
 - wildcards C-24
 - wire load model 3-12
 - wiring pins, power gating C-28
 - wiring power-gating pins C-28
 - worse negative slack 8-15
 - write_script command 6-12, 6-15, 7-59
 - writing a design to disk 1-7
 - writing from simulation
 - toggle_report 4-10
- ## Z
- zero-delay model 6-4
 - zero-delay simulation
 - creating a clock for 5-10