

Library Compiler™

User Guide

Version L-2016.06-SP2, September 2016

SYNOPSYS®

Copyright Notice and Proprietary Information

©2016 Synopsys, Inc. All rights reserved. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

Destination Control Statement

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

Disclaimer

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

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at <http://www.synopsys.com/Company/Pages/Trademarks.aspx>.
All other product or company names may be trademarks of their respective owners.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

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

Copyright Notice for the Command-Line Editing Feature

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

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

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement:

This product includes software developed by the University of California, Berkeley and its contributors.

4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

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

Copyright Notice for the Line-Editing Library

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

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

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

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

Copyright Notice for the Nonlinear Optimization Library

© 2011 by Massachusetts Institute of Technology.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

1. The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
2. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Contents

About This Guide	xxxvi
Customer Support.....	xxxix
1. Introduction to the Library Compiler Tool	
The Role of the Library Compiler Tool	1-2
Creating Libraries	1-2
Technology Libraries.....	1-2
Compiled Libraries and the Design Compiler Tool	1-5
2. Library Development Procedure	
Library Development Overview.....	2-2
Task 1: Understanding Language Concepts	2-2
References	2-3
Basic Syntax Rules	2-3
Group Statements	2-3
Attribute Statements	2-3
Task 2: Creating a Library.....	2-4
References	2-4
Procedure for Creating a Library	2-4
Determining the Technology	2-4
Selecting a Delay Model	2-4
Determining Where This Library Is Used.....	2-4
Describing the Library	2-5

Naming the Library	2-5
Defining the library Group.....	2-5
Specifying library Group Attributes	2-5
Task 3: Describing the Environment.....	2-6
References	2-6
Procedure	2-6
Modeling Delay Calculation Scaling Factors	2-7
Defining Default Attributes	2-7
Defining Operating Conditions.....	2-7
Modeling the Wire Load	2-9
Specifying Values for the 3-D Wire Delay Lookup Table.	2-9
Task 4: Defining the Cells	2-10
References	2-11
Procedure	2-11
Defining Core Cells	2-12
Referring to the Datasheet	2-12
Specifying the Cell Name	2-15
Listing the Pins	2-15
Determining Pin Functions	2-16
Describing Timing	2-18
Defining I/O Pads.	2-20
Identifying Pad Cells.....	2-21
Describing Multicell Pads.....	2-21
Defining Units for Pad Cells	2-22
Identifying Area Resources.....	2-22
Describing Input Pads	2-22
Describing Output Pads	2-23
Modeling Wire Load	2-23
Task 5: Describing Application-Specific Data	2-24
References	2-24
Procedure for Describing Application-Specific Data	2-24
Setting Design Rules	2-24
Modeling for Testable Circuits	2-25
Modeling for Power	2-25
Leakage Power	2-26
Short-Circuit Power.....	2-26
Switching Power	2-26
Technology Library With Delay Example	2-27

Task 6: Compiling the Library	2-34
Procedure	2-34
References	2-34
Task 7: Managing the Library	2-34
References	2-34
Procedure	2-34
3. Using the Library Compiler Shell	
About the Shell Interface	3-3
Starting the Command Interface	3-3
Understanding the Command Log	3-4
Using Setup Files	3-5
Using <code>lc_shell</code> Commands to Compile a Source File	3-5
Support for gzip Format	3-6
Interrupting Commands	3-6
Leaving the Command Interface	3-6
Describing Library Files and Memory Files	3-7
Using <code>search_path</code> and Path Names	3-8
<code>lc_shell</code> Command Syntax	3-9
Redirecting Command Output	3-10
Using Commands	3-10
Getting Help Information	3-10
Reading In Libraries	3-11
Listing Libraries	3-12
Writing Library Files	3-12
Generating Hierarchical .db Designs	3-13
Verifying Hierarchical .db Designs	3-14
Loading Compiled Libraries	3-14
Adding Library-Level Information to Existing Libraries	3-15
Updating Attribute Values	3-15
Accessing Library Information	3-15
Accessing Simple Attribute Values	3-16
Reporting Library Information	3-16

Library Scaling	3-18
Library Scaling Flow	3-19
Scaling Library Groups	3-21
Commands in the Library Scaling Flow	3-23
create_scaling_lib_group Command	3-23
check_library Command	3-23
write_scaling_lib_group Command	3-23
Scaling Library Flow Examples	3-24
Understanding the write_scaling_lib_group Output	3-26
Removing Libraries From Memory	3-28
Using Shell Commands	3-29
Command Aliases	3-29
unalias Command	3-30
Listing Previously Entered Commands	3-31
Reexecuting Commands	3-32
Shell Commands	3-33
Command Scripts	3-34
Using Control Flow Commands	3-35
Using the Wildcard Character	3-39
Using Variables	3-39
Setting Variable Values	3-40
Creating Variables	3-40
Listing Variable Values	3-40
Removing Variables	3-40
Productions	3-41
Syntax	3-42
4. Using Library Compiler Syntax	
Library File Names	4-2
Library, Cell, and Pin Names	4-2
Statements	4-2
Group Statements	4-3
Attribute Statements	4-3
Simple Attributes	4-3
Complex Attributes	4-4

Define Statements	4-4
Defining New Groups	4-4
Defining New Attributes	4-5
Comments.....	4-6
Units of Measure.....	4-6
Library Example	4-7
5. Building a Technology Library	
Creating Library Groups	5-2
Syntax	5-2
library Group	5-3
Using General Library Attributes.....	5-3
technology Attribute.....	5-3
delay_model Attribute	5-4
bus_naming_style Attribute	5-4
Documenting Libraries	5-5
date Attribute	5-5
revision Attribute	5-5
comment Attribute	5-5
Delay and Slew Attributes.....	5-6
input_threshold_pct_fall Simple Attribute	5-7
input_threshold_pct_rise Simple Attribute	5-8
output_threshold_pct_fall Simple Attribute	5-8
output_threshold_pct_rise Simple Attribute	5-9
slew_derate_from_library Simple Attribute	5-9
slew_lower_threshold_pct_fall Simple Attribute	5-10
slew_lower_threshold_pct_rise Simple Attribute	5-10
slew_upper_threshold_pct_fall Simple Attribute	5-11
slew_upper_threshold_pct_rise Simple Attribute	5-11
Defining Units	5-12
time_unit Attribute	5-12
voltage_unit Attribute	5-12
current_unit Attribute	5-13
pulling_resistance_unit Attribute	5-13

capacitive_load_unit Attribute	5-14
leakage_power_unit Attribute	5-14
Describing Pads	5-14
input_voltage Group	5-15
output_voltage Group	5-15
Describing Power	5-16
power_lut_template Group	5-16
Template Variables	5-17
Template Breakpoints	5-17
Using Processing Attributes	5-18
Setting Minimum Cell Requirements	5-19
Setting CMOS Default Attributes	5-19

6. Building Environments

Library-Level Default Attributes	6-2
Setting Default Cell Attributes	6-2
default_cell_leakage_power Simple Attribute	6-2
Setting Default Pin Attributes	6-2
Setting Wire Load Defaults	6-3
default_wire_load Attribute	6-3
Synchronizing Design and Model Modes	6-3
default_wire_load_capacitance Attribute	6-4
default_wire_load_resistance Attribute	6-4
default_wire_load_area Attribute	6-4
Setting Other Environment Defaults	6-4
default_operating_conditions Attribute	6-4
default_connection_class Attribute	6-5
Examples of Library-Level Default Attributes	6-5
em_temp_degradation_factor Attribute	6-6
Defining Operating Conditions	6-6
operating_conditions Group	6-6
Defining Timing Ranges at Runtime	6-8
Defining Wire Load Groups	6-8
wire_load Group	6-8
Calculating Wire Area	6-12

wire_load_table Group	6-14
Selecting Wire Load Groups Automatically	6-15
wire_load_from_area Attribute	6-15
Specifying Default Wire Load Settings	6-16
Specifying Delay Scaling Attributes	6-17
Calculating Delay Factors	6-17
Calculating Voltage Delay Factors	6-18
Calculating Temperature Delay Factors	6-18
Assigning Process Delay Factors	6-19
Setting Combined Scaling Factor	6-20
Pin and Wire Capacitance Factors	6-20
Scaling Factors Associated With the Nonlinear Delay Model	6-21
Specifying Nonlinear Delay Tables	6-21
7. Defining Core Cells	
Defining cell Groups	7-2
cell Group	7-2
area Attribute	7-3
cell_footprint Attribute	7-4
clock_gating_integrated_cell Attribute	7-4
Setting Pin Attributes for an Integrated Cell	7-6
Setting Timing for an Integrated Cell	7-7
contention_condition Attribute	7-7
dont_fault Attribute	7-7
dont_touch Attribute	7-8
dont_use Attribute	7-8
is_clock_gating_cell Attribute	7-8
is_macro_cell Attribute	7-9
is_memory_cell Attribute	7-9
map_only Attribute	7-9
pad_cell Attribute	7-9
pad_type Attribute	7-10
pin_equal Attribute	7-10
pin_opposite Attribute	7-10
preferred Attribute	7-11
use_for_size_only Attribute	7-11

short Attribute	7-12
type Group.....	7-13
cell Group Example	7-13
mode_definition Group	7-15
Group Statement.....	7-15
mode_value Group	7-15
Defining pin Groups	7-17
pin Group	7-17
General pin Group Attributes.....	7-19
capacitance Attribute	7-19
clock_gate_clock_pin Attribute	7-20
clock_gate_enable_pin Attribute.....	7-20
clock_gate_obs_pin Attribute	7-21
clock_gate_out_pin Attribute	7-21
clock_gate_test_pin Attribute	7-21
complementary_pin Simple Attribute	7-22
connection_class Simple Attribute	7-23
direction Attribute	7-24
dont_fault Attribute	7-25
driver_type Attribute	7-25
fall_capacitance Simple Attribute	7-30
fault_model Simple Attribute	7-31
inverted_output Attribute.....	7-32
is_analog Attribute	7-33
pin_func_type Attribute	7-33
rise_capacitance Simple Attribute	7-34
steady_state_resistance Attributes.....	7-34
test_output_only Attribute.....	7-35
Describing Design Rule Checks	7-35
fanout_load Attribute	7-36
max_fanout Attribute	7-36
min_fanout Attribute	7-37
max_transition Attribute	7-38
max_trans Group	7-39
max_capacitance Attribute	7-41
max_cap Group	7-41
min_capacitance Attribute	7-43
cell_degradation Group	7-44
Assigning Values to Lookup Tables	7-45
Describing Clocks	7-45

clock Attribute	7-46
min_period Attribute	7-46
min_pulse_width_high and min_pulse_width_low Attributes	7-46
Describing Clock Pin Functions	7-47
internal_node Attribute	7-51
Describing Sequential Devices	7-51
prefer_tied Attribute	7-51
Describing Pad Pins	7-52
Slew-Rate Control Attributes	7-56
Defining Bused Pins	7-58
type Group	7-58
bus Group	7-60
bus_type Attribute	7-60
Pin Attributes and Groups	7-60
Example Bus Description	7-61
Defining Signal Bundles	7-65
bundle Group	7-65
members Attribute	7-65
pin Attributes	7-66
Defining Layout-Related Multibit Attributes	7-67
Defining Multiplexers	7-68
Library Requirements	7-69
Defining Decoupling Capacitor Cells, Filler Cells, and Tap Cells	7-70
Syntax	7-71
Cell-Level Attributes	7-71
is_decap_cell	7-71
is_filler_cell	7-71
is_tap_cell	7-71

8. Defining Sequential Cells

Using Sequential Cell Syntax	8-2
Describing a Flip-Flop	8-3
Using the ff Group	8-3
clocked_on and clocked_on_also Attributes	8-4
next_state Attribute	8-4

nextstate_type Attribute	8-5
clear Attribute	8-5
preset Attribute	8-5
clear_preset_var1 Attribute	8-5
clear_preset_var2 Attribute	8-6
power_down_function Attribute	8-6
ff Group Examples	8-7
Describing a Single-Stage Flip-Flop	8-8
Describing a Master-Slave Flip-Flop	8-10
Using the function Attribute	8-11
Describing a Multibit Flip-Flop	8-12
Describing a Latch	8-15
latch Group	8-15
enable and data_in Attributes	8-16
data_in_type Attribute	8-16
clear Attribute	8-17
preset Attribute	8-17
clear_preset_var1 Attribute	8-17
clear_preset_var2 Attribute	8-18
Determining a Latch Cell's Internal State	8-19
Describing a Multibit Latch	8-19
latch_bank Group	8-19
Describing Sequential Cells With the Statetable Format	8-24
statetable Group	8-26
Using Shortcuts	8-28
Partitioning the Cell Into a Model	8-30
Defining an Output pin Group	8-31
state_function Attribute	8-31
internal_node Attribute	8-32
input_map Attribute	8-32
inverted_output Attribute	8-34
Internal Pin Type	8-34
Statetable Checks	8-35
Timing	8-36
Recommended Statetable Practices	8-36
Determining a Complex Sequential Cell's Internal State	8-38
Flip-Flop and Latch Examples	8-38

Cell Description Examples	8-43
9. Defining Test Cells	
Describing a Scan Cell	9-2
test_cell Group	9-2
Pins in the test_cell Group	9-3
test_output_only Attribute	9-6
signal_type Attribute	9-7
Describing a Multibit Scan Cell	9-7
Multibit Scan Cell With Parallel Scan Chain	9-8
Example	9-10
Multibit Scan Cell With Internal Scan Chain	9-14
Attributes Defined in the bus or bundle Group	9-16
Examples	9-18
Using Sequential Mapping-Based Scan Insertion	9-25
Scan Cell Modeling Examples	9-26
Simple Multiplexed D Flip-Flop	9-26
Multibit Cells With Multiplexed D Flip-Flop and Enable	9-28
LSSD Scan Cell	9-33
Scan-Enabled LSSD Cell	9-36
Functional Model of the Scan-Enabled LSSD Cell	9-37
Timing Model of the Scan-Enabled LSSD Cell	9-38
Scan-Enabled LSSD Cell Model Example	9-39
Scan-Enabled LSSD Cell With Asynchronous Inputs	9-42
Multibit Scan-Enabled LSSD Cell	9-44
Clocked-Scan Test Cell	9-46
Scan D Flip-Flop With Auxiliary Clock	9-48
10. Nonlinear Delay Model	
Total Delay Equation	10-2
Cell Delay	10-3
Propagation Delay	10-4
Transition Delay	10-4
Connect Delay	10-4
CMOS Nonlinear Delay Model Calculation	10-5

Process, Voltage, and Temperature Scaling	10-7
---	------

11. Timing Arcs

Understanding Timing Arcs	11-3
Combinational Timing Arcs	11-3
Sequential Timing Arcs	11-4
Modeling Method Alternatives	11-4
Defining the timing Group	11-5
Naming Timing Arcs Using the timing Group	11-6
Timing Arc Between a Single Pin and a Single Related Pin	11-7
Timing Arcs Between a Pin and Multiple Related Pins	11-7
Timing Arcs Between a Bundle and a Single Related Pin	11-8
Timing Arcs Between a Bundle and Multiple Related Pins	11-9
Timing Arcs Between a Bus and a Single Related Pin	11-11
Timing Arcs Between a Bus and Multiple Related Pins	11-12
Delay Model	11-14
delay_model Attribute	11-14
Defining the NLDM Template	11-14
Creating Lookup Tables	11-18
timing Group Attributes	11-19
related_pin Simple Attribute	11-20
related_bus_pins Simple Attribute	11-21
timing_sense Simple Attribute	11-21
timing_type Simple Attribute	11-22
mode Complex Attribute	11-28
Describing Three-State Timing Arcs	11-33
Describing Three-State-Disable Timing Arcs	11-33
Describing Three-State-Enable Timing Arcs	11-34
Describing Edge-Sensitive Timing Arcs	11-35
Describing Preset and Clear Timing Arcs	11-36
Describing Preset Arcs	11-36
Describing Clear Arcs	11-37
Describing Clock Insertion Delay	11-38
Describing Intrinsic Delay	11-39
Describing Transition Delay	11-40

Defining Delay Arcs With Lookup Tables.....	11-40
Modeling Transition Time Degradation	11-47
Modeling Load Dependency.....	11-50
Describing Slope Sensitivity	11-52
Describing State-Dependent Delays.....	11-52
when Simple Attribute	11-53
sdf_cond Simple Attribute	11-55
Setting Setup and Hold Constraints	11-57
rise_constraint and fall_constraint Groups	11-59
Identifying Interdependent Setup and Hold Constraints	11-61
Setting Nonsequential Timing Constraints	11-61
Setting Recovery and Removal Timing Constraints	11-62
Recovery Constraints	11-63
Removal Constraint	11-65
Setting No-Change Timing Constraints	11-66
In the CMOS Scalable Polynomial Delay Model	11-69
Setting Skew Constraints	11-70
Setting Conditional Timing Constraints.....	11-72
when and sdf_cond Simple Attributes	11-72
when_start Simple Attribute	11-73
sdf_cond_start Simple Attribute	11-73
when_end Simple Attribute	11-73
sdf_cond_end Simple Attribute	11-74
sdf_edges Simple Attribute	11-74
min_pulse_width Group.....	11-74
min_pulse_width Example	11-74
constraint_high and constraint_low Simple Attributes	11-75
when and sdf_cond Simple Attributes	11-75
minimum_period Group.....	11-75
minimum_period Example	11-75
constraint Simple Attribute	11-75
when and sdf_cond Simple Attributes	11-75
min_pulse_width and minimum_period Example	11-76
Checking min_pulse_width and minimum_period Constraints	11-77

Using Conditional Attributes With No-Change Constraints	11-78
Generating an SDF File	11-79
Impossible Transitions	11-80
Examples of NLDM Libraries	11-81
Library With Timing Constraints	11-81
CMOS Scalable Polynomial Delay Model	11-84
Library With Clock Insertion Delay	11-87
Describing a Transparent Latch Clock Model	11-88
Checking Timing and Nonlinear Delay Data	11-90
Driver Waveform Support	11-90
Library-Level Tables, Attributes, and Variables	11-92
normalized_voltage Variable	11-92
normalized_driver_waveform Group	11-92
driver_waveform_name Attribute	11-92
Cell-Level Attributes	11-93
driver_waveform Attribute	11-93
driver_waveform_rise and driver_waveform_fall Attributes	11-93
Pin-Level Attributes	11-94
driver_waveform Attribute	11-94
driver_waveform_rise and driver_waveform_fall Attributes	11-94
Checking Driver Waveform Data	11-94
Driver Waveform Example	11-94
Sensitization Support	11-96
sensitization Group	11-96
pin_names Attribute	11-97
vector Attribute	11-97
Cell-Level Attributes	11-97
sensitization_master Attribute	11-98
pin_name_map Attribute	11-98
timing Group Attributes	11-98
sensitization_master Attribute	11-98
pin_name_map Attribute	11-98
wave_rise and wave_fall Attributes	11-99
wave_rise_sampling_index and wave_fall_sampling_index Attributes	11-100
wave_rise_time_interval and wave_fall_time_interval Attributes	11-100
timing Group Syntax	11-101

Checking Sensitization Data	11-102
NAND Cell Example	11-102
Complex Macro Cell Example	11-103
Phase-Locked Loop Support	11-105
Phase-Locked Loop Syntax	11-106
Cell-Level Attribute	11-107
is_pll_cell Attribute	11-107
Pin-Level Attributes	11-107
is_pll_reference_pin Attribute	11-107
is_pll_feedback_pin Attribute	11-107
is_pll_output_pin Attribute	11-107
Phase-Locked Loop Example	11-108
Checking Phase-Locked Loop Libraries	11-109
12. Modeling Power and Electromigration	
Modeling Power Terminology	12-2
Static Power	12-2
Dynamic Power	12-2
Internal Power	12-2
Switching Power	12-3
Switching Activity	12-4
Modeling Cells for Power	12-4
Modeling for Leakage Power	12-6
Representing Leakage Power Information	12-6
cell_leakage_power Simple Attribute	12-6
Using the leakage_power Group for a Single Value	12-7
mode Complex Attribute	12-7
when Simple Attribute	12-8
value Simple Attribute	12-10
leakage_power_unit Simple Attribute	12-10
default_cell_leakage_power Simple Attribute	12-11
Example	12-11
Threshold Voltage Modeling	12-14
Modeling for Internal and Switching Power	12-15
Modeling Internal Power Lookup Tables	12-16

Clock Pin Power	12-18
Output Pin Power	12-18
Calculating Switching Power	12-20
Representing Internal Power Information	12-21
Specifying the Power Model	12-21
Using Lookup Table Templates	12-21
power_lut_template Group	12-22
Scalar power_lut_template Group	12-24
Defining Internal Power Groups	12-24
Naming Power Relationships, Using the <code>internal_power</code> Group	12-24
Power Relationship Between a Single Pin and a Single Related Pin	12-25
Power Relationships Between a Single Pin and Multiple Related Pins	12-25
Power Relationships Between a Bundle and a Single Related Pin	12-26
Power Relationships Between a Bundle and Multiple Related Pins	12-27
Power Relationships Between a Bus and a Single Related Pin	12-29
Power Relationships Between a Bus and Multiple Related Pins	12-30
Power Relationships Between a Bus and Related Bus Pins	12-32
<code>internal_power</code> Group	12-33
<code>equal_or_opposite_output</code> Simple Attribute	12-34
<code>falling_together_group</code> Simple Attribute	12-34
<code>related_pin</code> Simple Attribute	12-35
<code>rising_together_group</code> Simple Attribute	12-36
<code>switching_interval</code> Simple Attribute	12-36
<code>switching_together_group</code> Simple Attribute	12-37
<code>when</code> Simple Attribute	12-38
<code>mode</code> Complex Attribute	12-38
<code>fall_power</code> Group	12-40
<code>power</code> Group	12-41
<code>rise_power</code> Group	12-42
Internal Power Examples	12-45
One-Dimensional Power Lookup Table	12-45
Two-Dimensional Power Lookup Table	12-47
Three-Dimensional Power Lookup Table	12-48
Multiple Power Supply Library Requirements	12-49
Modeling Libraries With Integrated Clock-Gating Cells	12-50
What Clock Gating Does	12-50
Looking at a Gated Clock	12-52

Using an Integrated Clock-Gating Cell	12-53
Setting Pin Attributes for an Integrated Cell.....	12-53
clock_gate_clock_pin Attribute	12-54
clock_gate_enable_pin Simple Attribute	12-54
clock_gate_test_pin Attribute	12-55
clock_gate_obs_pin Attribute	12-55
clock_gate_out_pin Attribute	12-55
Clock-Gating Timing Considerations	12-56
Timing Considerations for Integrated Cells.....	12-57
Integrated Clock-Gating Cell Example	12-58
Modeling Electromigration	12-60
Overview	12-61
Controlling Electromigration.....	12-62
em_lut_template Group	12-63
electromigration Group	12-65

13. Advanced Low-Power Modeling

Power and Ground (PG) Pins	13-3
Partial PG Pin Cell Modeling	13-3
Special Partial PG Pin Cells	13-4
Supported Attributes.....	13-5
Partial PG Pin Cell Example.....	13-6
Partial PG Pin Cell Checks.....	13-7
PG Pin Syntax	13-7
Library-Level Attributes	13-8
voltage_map Complex Attribute	13-8
default_operating_conditions Simple Attribute	13-8
Cell-Level Attributes.....	13-8
pg_pin Group	13-9
is_pad Simple Attribute.....	13-9
voltage_name Simple Attribute.....	13-9
pg_type Simple Attribute.....	13-9
physical_connection Simple Attribute.....	13-10
related_bias_pin Attribute.....	13-11
user_pg_type Simple Attribute	13-11
Attributes Specifying Bias PG Pins With Insulated Substrate Wells	13-12
Pin-Level Attributes	13-12
power_down_function Attribute	13-12
related_power_pin and related_ground_pin Attributes	13-13

output_signal_level_low and output_signal_level_high Attributes	13-13
input_signal_level_low and input_signal_level_high Attributes	13-13
related_pg_pin Attribute	13-13
Naming Conventions for Power and Ground Pins in Technology Libraries	13-14
PG Pin Library Checks	13-14
Standard Cell With One Power and Ground Pin Example	13-15
Inverter With Substrate-Bias Pins Example	13-17
Defining Power Data for Multiple-Rail Cells	13-19
Insulated Bias Modeling	13-22
Feedthrough Signal Pin Modeling	13-26
Multipin Feedthroughs	13-26
short Attribute	13-27
Single-Pin Feedthrough	13-29
Silicon-on-Insulator (SOI) Cell Modeling	13-31
Library-Level Attribute	13-33
is_soi Attribute	13-33
Cell-Level Attribute	13-33
is_soi Attribute	13-33
SOI Cell Checks	13-33
Level-Shifter Cells in a Multivoltage Design	13-33
Operating Voltages	13-34
Level Shifter Functionality	13-34
Basic Level-Shifter Syntax	13-34
Cell-Level Attributes	13-35
is_level_shifter Attribute	13-35
level_shifter_type Attribute	13-36
input_voltage_range Attribute	13-36
output_voltage_range Attribute	13-36
Pin-Level Attributes	13-37
std_cell_main_rail Attribute	13-37
level_shifter_data_pin Attribute	13-37
level_shifter_enable_pin Attribute	13-37
input_voltage_range and output_voltage_range Attributes	13-38
input_signal_level Attribute	13-38
power_down_function Attribute	13-38
alive_during_power_up Attribute	13-38
Clamping Enable Level-Shifter Cell Outputs	13-38
Pin-Level Attributes	13-39

Level-Shifter Cell With PG Pin Not Connected to Switching Domains	13-40
Syntax	13-40
Level Shifter Modeling Examples	13-42
Simple Buffer Type Low-to-High Level Shifter	13-42
Simple Buffer Type High-to-Low Level Shifter	13-45
Multibit Level-Shifter Cell	13-47
Overdrive Level-Shifter Cell	13-49
Level-Shifter Cell With Virtual Bias Pins	13-51
Enable Level-Shifter Cell	13-52
Level-Shifter Cell Not Powered by Switching Power Domains	13-56
Level-Shifter Library Checks	13-60
Isolation Cell Modeling	13-60
Cell-Level Attribute	13-61
is_isolation_cell Attribute	13-61
Pin-Level Attributes	13-61
isolation_cell_data_pin Attribute	13-61
isolation_cell_enable_pin Attribute	13-61
power_down_function Attribute	13-61
permit_power_down Attribute	13-62
alive_during_partial_power_down Attribute	13-62
alive_during_power_up Attribute	13-62
Attribute Dependencies	13-62
Isolation Cell Examples	13-63
Multibit Isolation Cell	13-65
Clock-Isolation Cell Modeling	13-67
Cell-Level Attribute	13-68
Pin-Level Attributes	13-68
Clock Isolation Cell Examples	13-69
Clamping Isolation Cell Output Pins	13-71
Pin-Level Attributes	13-72
Example of Clamping in Isolation Cell	13-73
Isolation Cells With Multiple Control Pins	13-74
Isolation Cell Library Checks	13-76
Switch Cell Modeling	13-76
Coarse-Grain Switch Cells	13-77
Coarse-Grain Switch Cell Syntax	13-77
Library-Level Group	13-79
Cell-Level Attribute and Group	13-80
Pin-Level Attributes	13-80

pg_pin Group	13-82
Fine-Grained Switch Support for Macro Cells	13-82
Macro Cell With Fine-Grained Switch Syntax	13-82
Cell-Level Attributes	13-83
pg_pin Group	13-83
Switch-Cell Modeling Examples	13-83
Simple Coarse-Grain Header Switch Cell.	13-83
Complex Coarse-Grain Header Switch Cell	13-85
Complex Coarse-Grain Switch Cell With an Internal Switch Pin	13-88
Complex Coarse-Grain Switch Cell With Parallel Switches	13-90
Switch Cell Library Checks	13-93
Retention Cell Modeling	13-93
Modes of Operation	13-95
Retention Cell Modeling Syntax.	13-95
Cell-Level Attributes, Groups, and Variables	13-98
retention_cell Simple Attribute	13-98
ff, latch, ff_bank, and latch_bank Groups	13-98
retention_condition Group	13-98
clock_condition Group	13-99
preset_condition and clear_condition Groups	13-100
reference_pin_names Variable	13-100
variable1 and variable2 Variables.	13-101
bits Variable	13-101
Pin-Level Attributes	13-101
retention_pin Complex Attribute	13-101
function Attribute.	13-102
reference_input Attribute	13-102
save_action and restore_action Attributes	13-102
restore_edge_type Attribute	13-103
save_condition and restore_condition Attributes	13-104
Retention Cell Model Examples.	13-104
Zero-Pin Retention Cell Example	13-121
Multibit Retention Scan Cell Examples.	13-124
Retention Cell Library Checks	13-131
Always-On Cell Modeling	13-131
Always-On Cell Syntax	13-132
always_on Simple Attribute	13-132
Always-On Simple Buffer Example	13-132
Always-On Cell Library Checks	13-134

Macro Cell Modeling	13-134
Macro Cell Isolation Modeling	13-134
Pin-Level Attributes	13-137
I/O Pad Cell Internal Isolation Modeling	13-138
Pin-Level Attributes	13-140
I/O Pad Cell Isolation Checks	13-141
Modeling Macro Cells With Internal PG Pins	13-141
Low Power Macro Cell Modeling	13-144
Macro Cell With Fine-Grained Internal Power Switches	13-146
Macro Cell With an Always-On Pin Example	13-148
Macro Cell Isolation Checks	13-149
Modeling Antenna Diodes	13-149
Antenna-Diode Cell Modeling	13-149
Cell-Level Attribute	13-150
Pin-Level Attributes	13-151
Antenna-Diode Cell Modeling Example	13-151
Modeling Cells With Built-In Antenna-Diode Ports	13-152
Pin-Level Attributes	13-152
Antenna-Diode Pin Modeling Example	13-153
Antenna-Diode Cell Checks	13-154

14. Composite Current Source Modeling

Modeling Cells With Composite Current Source Information	14-2
Representing Composite Current Source Driver Information	14-2
Composite Current Source Lookup Tables	14-2
Defining the output_current_template Group	14-2
output_current_template Syntax	14-2
Template Variables for CCS Driver Models	14-3
output_current_template Example	14-3
Defining the Lookup Table Output Current Groups	14-3
output_current_rise Syntax	14-3
vector Group	14-3
reference_time Simple Attribute	14-4
Template Variables for CCS Driver Models	14-4
vector Group Example	14-4
Checking CCS Driver Models	14-5
Representing Composite Current Source Receiver Information	14-5

Comparison Between Two-Segment and Multisegment Receiver Models	14-5
Two-Segment Receiver Capacitance Model	14-6
Syntax	14-7
Defining the receiver_capacitance Group at the Pin Level	14-8
Defining the lu_table_template Group	14-8
Conditional Data Modeling	14-9
Examples	14-9
Checking the receiver_capacitance Group	14-14
Defining the Receiver Capacitance Groups at the Timing Level	14-14
Conditional Data Modeling	14-14
Defining the lu_table_template Group	14-14
Timing-Level receiver_capacitance Example	14-15
Checking the Timing-Level receiver_capacitance Group	14-16
Multisegment Receiver Capacitance Model	14-16
Library-Level Groups and Attributes	14-17
lu_table_template Group	14-17
receiver_capacitance_rise_threshold_pct and receiver_capacitance_fall_threshold_pct Attributes	14-17
Pin and Timing Level Groups	14-18
receiver_capacitance Group	14-18
Conditional Data Modeling	14-19
Example	14-19
CCS Retain Arc Support	14-21
CCS Retain Arc Syntax	14-22
ccs_retain_rise and ccs_retain_fall Groups	14-23
vector Group	14-23
reference_time Attribute	14-24
Compact CCS Retain Arc Syntax	14-24
compact_ccs_retain_rise and compact_ccs_retain_fall Groups	14-25
base_curves_group Attribute	14-25
index_1, index_2, and index_3 Attributes	14-25
values Attribute	14-26
Composite Current Source Driver and Receiver Model Example	14-26
Checking CCS Library Models	14-28

15. Advanced Composite Current Source Modeling

Modeling Cells With Advanced Composite Current Source Information	15-2
---	------

Compact CCS Timing Model	15-2
Modeling With CCS Timing Base Curves	15-2
Compact CCS Timing Model Syntax	15-4
base_curves Group	15-6
compact_lut_template Group	15-6
compact_ccs_{rise fall} Group	15-7
Checking the compact_lut_template Group	15-8
CCS Timing Library Example	15-8

16. Composite Current Source Signal Integrity Modeling

CCS Signal Integrity Modeling Overview	16-2
Compiling a Library With CCS Signal Integrity Information	16-2
CCS Signal Integrity Modeling Syntax	16-2
Library-Level Groups and Attributes	16-7
lu_table_template Group	16-7
variable_1, variable_2, variable_3, and variable_4 Attributes	16-7
Pin-Level Groups and Attributes	16-8
ccsn_first_stage and ccsn_last_stage Groups	16-8
is_needed Attribute	16-8
is_inverting Attribute	16-9
stage_type Attribute	16-9
miller_cap_rise and miller_cap_fall Attributes	16-9
output_signal_level and input_signal_level Attributes	16-9
dc_current Group	16-10
output_voltage_rise and output_voltage_fall Groups	16-10
propagated_noise_high and propagated_noise_low Groups	16-10
when Attribute	16-11
Checking CCS Signal Integrity Models	16-11
CCS Noise Library Example	16-11
Conditional Data Modeling in CCS Noise Models	16-14
when Attribute	16-15
mode Attribute	16-15
CCS Noise Modeling for Unbuffered Cells With a Pass Gate	16-17
Syntax for Unbuffered Output Latches	16-18
Pin-Level Attributes	16-19
is_unbuffered Attribute	16-19
has_pass_gate Attribute	16-19
ccsn_first_stage Group	16-20
is_pass_gate Attribute	16-20

CCS Noise Modeling for Multivoltage Designs	16-20
Referenced CCS Noise Modeling	16-23
Modeling Syntax	16-24
Cell-Level Attributes	16-25
driver_waveform Attribute	16-25
driver_waveform_rise and driver_waveform_fall Attributes	16-26
Pin-Level Groups	16-26
input_ccb and output_ccb Groups	16-26
Timing-Level Attributes	16-27
active_input_ccb Attribute	16-27
active_output_ccb Attribute	16-28
propagating_ccb Attribute	16-28
Examples	16-28

17. Composite Current Source Power Modeling

Composite Current Source Power Modeling	17-2
Cell Leakage Current	17-2
Checking Leakage Current Syntax	17-3
Gate Leakage Modeling in Leakage Current	17-3
gate_leakage Group	17-4
input_low_value Attribute	17-4
input_high_value Attribute	17-5
Checking Gate Leakage Current Syntax	17-5
Intrinsic Parasitic Models	17-5
Voltage-Dependent Intrinsic Parasitic Models	17-7
Library-Level Group	17-10
lu_table_template Group	17-10
Pin-Level Group	17-10
lut_values Group	17-10
Checking Intrinsic Parasitic Syntax	17-10
Parasitics Modeling in Macro Cells	17-11
total_capacitance Group	17-11
Parasitics Modeling Syntax	17-11
Checking Parasitic Model Syntax	17-11
Dynamic Power	17-11
Dynamic Power and Ground Current Table Syntax	17-12
Dynamic Power Modeling in Macro Cells	17-12
Checking Dynamic Power Model Syntax	17-14
Examples for CCS Dynamic Power for Macro Cells	17-14

Conditional Data Modeling for Dynamic Current Example	17-16
Checking Power and Ground Current Syntax	17-18
Dynamic Current Syntax	17-18
Checking Dynamic Current Syntax	17-19
Compact CCS Power Modeling	17-19
Compact CCS Power Syntax and Requirements	17-21
Library-Level Groups and Attributes	17-23
base_curves Group	17-23
compact_lut_template Group	17-24
Cell-Level Groups and Attributes	17-25
compact_ccs_power Group	17-25
Composite Current Source Dynamic Power Examples	17-26
Design Cell With a Single Output Example	17-26
Dense Table With Two Output Pins Example	17-27
Cross Type With More Than One Output Pin Example	17-27
Diagonal Type With More Than One Output Pin Example	17-29
18. On-Chip Variation (OCV) Modeling	
OCV Model Overview	18-2
Advanced OCV Modeling	18-2
Library-Level Groups and Attributes	18-4
ocv_table_template Group	18-4
ocv_derate Group	18-5
default_ocv_derate_group Attribute	18-6
ocv_arc_depth Attribute	18-6
Cell-Level Attribute	18-7
ocv_derate_group Attribute	18-7
Advanced OCV Library Example	18-7
Advanced OCV Checks	18-9
Parametric OCV Liberty Variation Format Modeling	18-9
Library-Level Groups and Attributes	18-12
lu_table_template Group	18-12
ocv_table_template Group	18-13
ocv_derate Group	18-14
default_ocv_derate_distance_group Attribute	18-15
Cell-Level Attribute	18-15
ocv_derate_distance_group Attribute	18-15

Arc-Level Groups and Attributes	18-15
ocv_sigma_cell_rise Group	18-15
ocv_sigma_cell_fall Group	18-16
ocv_sigma_rise_transition Group	18-16
ocv_sigma_fall_transition Group	18-16
sigma_type Attribute	18-17
ocv_sigma_rise_constraint Group	18-17
ocv_sigma_fall_constraint Group	18-17
Parametric OCV Library Example	18-18
Parametric OCV Checks	18-21

19. Defining Library Model Files

Model Files	19-2
Types of Model Files	19-2
Power-Aware Model File	19-3
Model File Syntax	19-4
Liberty Variation Format Advanced OCV Model Syntax	19-4
Liberty Variation Format Parametric OCV Model Syntax	19-5
CCS Power Model Syntax	19-7
Power-Aware Model Syntax	19-9
Reference Syntax	19-12
General	19-12
Reference Syntax For Specific Model Types	19-13
Framework Syntax For Power-Aware Model Files	19-15
Compiling Model Files	19-16
Compiling Power-Aware Model Files	19-17

20. Modeling Timing of Complex Macro Blocks

Interface Timing Specification	20-2
Sequential Cell Timing Without Interface Timing Specification	20-3
Supported Features of Interface Timing	20-3
Describing Blocks With Interface Timing	20-5
Using Library Compiler Syntax	20-5
Differences From Regular Cell Specification	20-5
Advanced Modeling Technology	20-5
model Group	20-6
model Group Example	20-7

generated_clock Group	20-7
Generated Clock Example	20-11
Similarities to Regular Cell Specification	20-11
Examples of Interface Timing Relationships	20-12
Interface Timing of a Complex Sequential Block	20-15
Using Blocks With Interface Timing in Synthesis	20-20
Interpreting Timing Relationships	20-20
Examples Using Interface Timing Specification	20-21
21. Defining I/O Pads	
Special Characteristics of I/O Pads	21-2
Identifying Pad Cells	21-2
Regular Pad Cells	21-2
Clock Pads	21-3
Pad Pin Attributes	21-3
Describing Multicell Pads	21-3
Auxiliary Pad Cells	21-4
Connecting Pins	21-5
Defining Units for Pad Cells	21-5
Units of Time	21-6
Capacitance	21-6
Resistance	21-6
Voltage	21-6
Current	21-7
Power	21-7
Describing Input Pads	21-7
Voltage Levels	21-7
Hysteresis	21-8
Describing Output Pads	21-8
Voltage Levels	21-8
Drive Current	21-9
Slew-Rate Control	21-9
Modeling Wire Load for Pads	21-11
Internal Isolation Support in I/O Pad Cell Models	21-12

Programmable Driver Type Support in I/O Pad Cell Models	21-12
Syntax	21-12
Programmable Driver Type Functions	21-13
Example	21-14
Reporting Pad Information	21-16
Pad Cell Examples	21-17
Input Pads	21-17
Output Pads	21-20
Bidirectional Pad	21-21
Components for Multicell Pads	21-23
Cell with contention_condition and x_function	21-25

Appendix A. Clock-Gating Integrated Cell Circuits

List and Description of Options	A-2
Schematics and Examples	A-7
latch_posedge Option	A-7
latch_posedge_precontrol Option	A-9
latch_posedge_postcontrol Option	A-11
latch_posedge_precontrol_obs Option	A-15
latch_posedge_postcontrol_obs Option	A-17
latch_negedge Option	A-18
latch_negedge_precontrol Option	A-20
latch_negedge_postcontrol Option	A-22
latch_negedge_precontrol_obs Option	A-24
latch_negedge_postcontrol_obs Option	A-26
none_posedge Option	A-28
none_posedge_control Option	A-29
none_posedge_control_obs Option	A-31
none_negedge Option	A-34
none_negedge_control Option	A-35
none_negedge_control_obs Option	A-37
Generic Integrated Clock-Gating Schematics	A-39
latch_posedgeactive_low	A-39
latch_posedgeactive_low_precontrol	A-40
latch_posedgeactive_low_postcontrol	A-40
latch_posedgeactive_low_precontrol_obs	A-41

latch_posedgeactivehigh_postcontrol_obs	A-41
none_posedgeactivehigh	A-41
none_posedgeactivehigh_control	A-42
none_posedgeactivehigh_control_obs	A-42
latch_nedgedeactivehigh	A-42
latch_nedgedeactivehigh_precontrol	A-43
latch_nedgedeactivehigh_postcontrol	A-43
latch_nedgedeactivehigh_precontrol_obs	A-44
latch_nedgedeactivehigh_postcontrol_obs	A-44
none_nedgedeactivehigh	A-44
none_nedgedeactivehigh_control	A-45
none_nedgedeactivehigh_control_obs	A-45
latch_posedge_invgclk	A-45
latch_posedge_precontrol_invgclk	A-46
latch_posedge_postcontrol_invgclk	A-46
latch_posedge_precontrol_obs_invgclk	A-46
latch_posedge_postcontrol_obs_invgclk	A-47
none_posedge_invgclk	A-47
none_posedge_control_invgclk	A-48
none_posedge_control_obs_invgclk	A-48
latch_nedgedeactivehigh_invgclk	A-48
latch_nedgedeactivehigh_precontrol_invgclk	A-49
latch_nedgedeactivehigh_postcontrol_invgclk	A-49
latch_nedgedeactivehigh_precontrol_obs_invgclk	A-49
latch_nedgedeactivehigh_postcontrol_obs_invgclk	A-50
none_nedgedeactivehigh_invgclk	A-50
none_nedgedeactivehigh_control_invgclk	A-50
none_nedgedeactivehigh_control_obs_invgclk	A-51
latch_posedgeactivehigh_invgclk	A-51
latch_posedgeactivehigh_precontrol_invgclk	A-51
latch_posedgeactivehigh_postcontrol_invgclk	A-52
latch_posedgeactivehigh_precontrol_obs_invgclk	A-52
latch_posedgeactivehigh_postcontrol_obs_invgclk	A-53
none_posedgeactivehigh_invgclk	A-53
none_posedgeactivehigh_control_invgclk	A-53
none_posedgeactivehigh_control_obs_invgclk	A-54
latch_nedgedeactivehigh_invgclk	A-54
latch_nedgedeactivehigh_precontrol_invgclk	A-54
latch_nedgedeactivehigh_postcontrol_invgclk	A-55
latch_nedgedeactivehigh_precontrol_obs_invgclk	A-55
latch_nedgedeactivehigh_postcontrol_obs_invgclk	A-55

none_nedgedeactive_low_invgclk	A-56
none_nedgedeactive_low_control_invgclk	A-56
none_nedgedeactive_low_control_obs_invgclk	A-56

Appendix B. Library Characterization Configuration

The char_config Group	B-2
Library Characterization Configuration Syntax	B-2
Common Characterization Attributes	B-5
driver_waveform Attribute	B-7
driver_waveform_rise and driver_waveform_fall Attributes	B-8
input_stimulus_transition Attribute	B-8
input_stimulus_interval Attribute	B-8
unrelated_output_net_capacitance Attribute	B-9
default_value_selection_method Attribute	B-9
default_value_selection_method_rise and default_value_selection_method_fall Attributes	B-9
merge_tolerance_abs and merge_tolerance_rel Attributes	B-10
merge_selection Attribute	B-11
Three-State Characterization Attributes	B-11
three_state_disable_measurement_method Attribute	B-11
three_state_disable_current_threshold_abs and three_state_disable_current_threshold_rel Attributes	B-12
three_state_disable_monitor_node Attribute	B-12
three_state_cap_add_to_load_index Attribute	B-12
CCS Timing Characterization Attributes	B-13
ccs_timing_segment_voltage_tolerance_rel Attribute	B-13
ccs_timing_delay_tolerance_rel Attribute	B-13
ccs_timing_voltage_margin_tolerance_rel Attribute	B-14
CCS Receiver Capacitance Attributes	B-14
Input-Capacitance Characterization Attributes	B-14
capacitance_voltage_lower_threshold_pct_rise and capacitance_voltage_lower_threshold_pct_fall Attributes	B-15
capacitance_voltage_upper_threshold_pct_rise and capacitance_voltage_upper_threshold_pct_fall Attributes	B-15

Preface

This preface includes the following sections:

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

About This Guide

The Library Compiler tool captures ASIC libraries and translates them into the Synopsys database format. This user guide describes how to use the Library Compiler software, the `lc_shell` commands, Liberty™ syntax, and explains how to build logic libraries and define cells. It also describes how to use the Liberty syntax to model timing, signal integrity, and power in logic libraries.

Library Compiler documentation also includes the following user guide and reference manual:

- *Library Quality Assurance System User Guide* provides information about reading and compiling libraries, generating library reports, and checking libraries.
 - *Synopsys Logic Library Reference Manual* provides the syntax of the group statements that identify the characteristics of logic libraries.
-

Audience

The target audience for the Library Compiler documentation suite comprises library designers, logic designers, and electronics engineers. Readers need a basic familiarity with optimization and analysis tools from Synopsys, as well as experience in reading manufacturers' specification sheets for ASIC components.

Related Publications

For additional information about the Library Compiler tool, see the documentation on Synopsys SolvNet® online support site at the following address:

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

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

- Design Compiler®
- Formality®
- Power Compiler™
- PrimeTime® and PrimeTime SI
- DFT Compiler

Release Notes

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

To see the *Library Compiler Release Notes*,

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

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

2. Select Library Compiler, and then select a release in the list that appears.

Conventions

The following conventions are used in Synopsys documentation.

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

Customer Support

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

Accessing SolvNet

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

To access SolvNet, go to the following address:

<https://solvnet.synopsys.com>

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

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

Contacting the Synopsys Technical Support Center

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

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

1

Introduction to the Library Compiler Tool

Library characterization data can be stored in a text file in the Liberty (.lib) format. The Library Compiler tool reads the description of an ASIC library from the .lib file and compiles the description into an internal database (.db file format).

The compiled library database can then be used for timing, power, and noise analysis with compatible synthesis, place and route, static timing analysis, and verification tools.

To access the Library Compiler tool, use the lc_shell command interface. For information about using the lc_shell command interface, see [Chapter 3, “Using the Library Compiler Shell.”](#) See the command man pages for lc_shell command descriptions.

This chapter describes:

- [The Role of the Library Compiler Tool](#)
- [Technology Libraries](#)
- [Compiled Libraries and the Design Compiler Tool](#)

The Role of the Library Compiler Tool

The role of the Library Compiler tool is to generate technology libraries that are used by Synopsys synthesis, place and route, static timing analysis, and verification tools.

The technology libraries contain information about the characteristics and functions of each component of an ASIC library. Information stored in the technology libraries consists of area, timing, function, power, and other technology information.

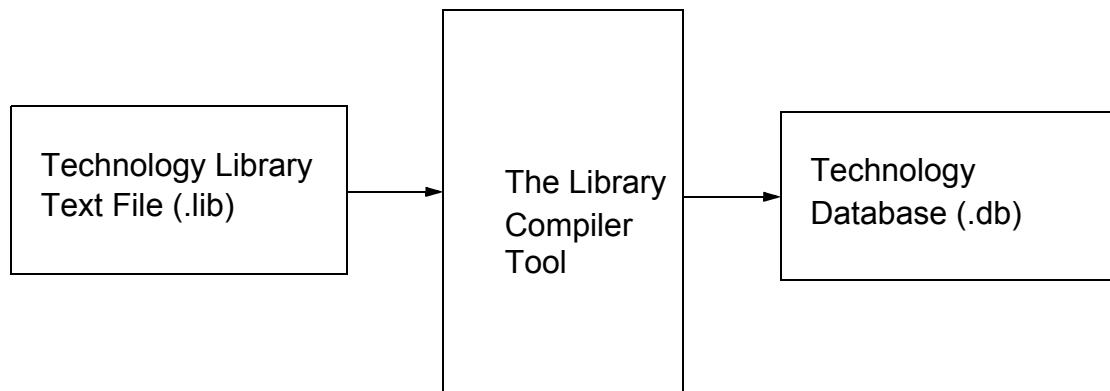
Creating Libraries

You can create technology libraries by writing ASCII text descriptions.

After you create the library files, you can use the Library Compiler tool to translate the files into the Synopsys internal database format (.db), which you can write out to a disk file and use repeatedly.

[Figure 1-1](#) shows the role of the Library Compiler tool.

Figure 1-1 The Role of the Library Compiler Tool



Technology Libraries

A technology library is a text file that contains the following information about the ASIC technology:

Structural information

Describes the connectivity of each cell, including pin, bus or bundle descriptions.

Functional information

Describes the logic function of the cell output pins for the Design Compiler tool to map the design logic to the ASIC technology.

Timing information

Describes the parameters for pin-to-pin timing relationships for delay calculation of each cell in the library. This information ensures accurate timing analysis and timing optimization of a design.

Environmental information

Describes the manufacturing process, operating temperature, supply voltage, and design layout, all of which directly affect the efficiency of every design.

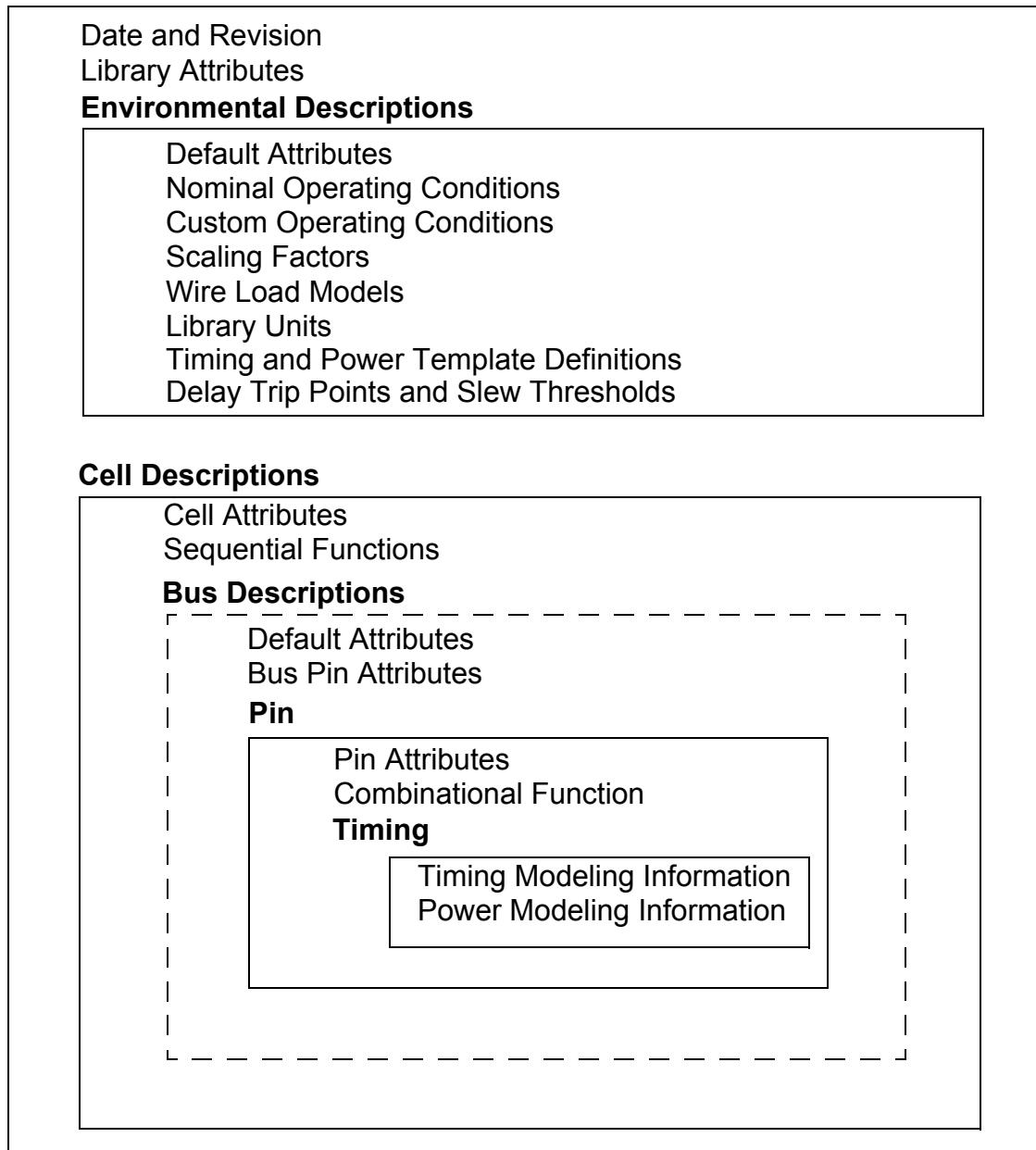
These variables and their effects are defined in the environment descriptions of the technology library. Environment descriptions include interconnect wire areas; wire capacitance and resistance; and the scaling factors for variations in process, temperature, and voltage.

[Figure 1-2](#) shows the structure of a technology library. A technology library contains cell and environment descriptions:

- Cell descriptions define each individual component in the ASIC technology, including cell, bus, pin, area, function, and timing.
- Environment descriptions contain information about the ASIC technology that is not unique to individual components. This section also contains information about the effects of operating conditions on the technology, values of the components of delay scaling equations, statistical data of interconnect estimation, and default cell attributes.

Figure 1-2 Technology Library Structure

Technology Library

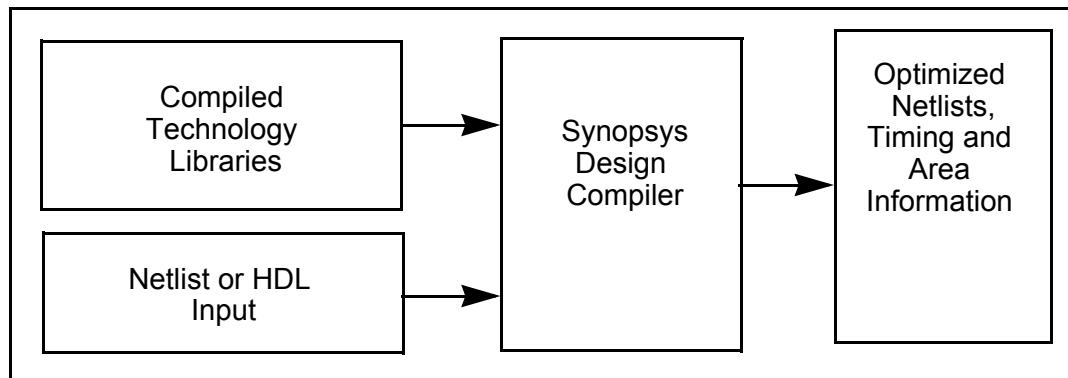


Compiled Libraries and the Design Compiler Tool

You can use a compiled technology library with the Design Compiler tool to generate designs, as shown in [Figure 1-3](#).

Compiled libraries are hardware independent. You can transfer libraries between different hardware platforms if the files are treated as binary files. To transfer the files, use the `ftp` or `rcp` command in binary mode.

Figure 1-3 Using Compiled Libraries With the Design Compiler Tool



2

Library Development Procedure

This chapter describes the general procedure for creating technology libraries and provides an overview of library development tasks.

This chapter includes the following sections:

- [Library Development Overview](#)
- [Task 1: Understanding Language Concepts](#)
- [Task 2: Creating a Library](#)
- [Task 3: Describing the Environment](#)
- [Task 4: Defining the Cells](#)
- [Task 5: Describing Application-Specific Data](#)
- [Task 6: Compiling the Library](#)
- [Task 7: Managing the Library](#)

Library Development Overview

A technology library describes the structure, function, timing, power, signal integrity, and environment of the ASIC technology being used. A technology library contains information used in these synthesis activities:

- Translation—functional information for each cell
- Optimization—area and timing information for each cell (including timing constraints on sequential cells)
- Design rule fixing—design rule constraints on cells

Library development consists of the following major activities:

1. Describing a library in text format (.lib).
2. Compiling the library into a binary form (.db).

To develop a library in text format and compile the library into binary format, perform the following tasks:

- [Task 1: Understanding Language Concepts](#)
- [Task 2: Creating a Library](#)
- [Task 3: Describing the Environment](#)
- [Task 4: Defining the Cells](#)
- [Task 5: Describing Application-Specific Data](#)
- [Task 6: Compiling the Library](#)
- [Task 7: Managing the Library](#)

Task 1: Understanding Language Concepts

Information in a library is contained in language statements. The three types of library statements are: group object, simple attribute, and complex attribute. Each statement type has its own syntax.

References

For information about the Library Compiler syntax, see

- [Chapter 4, “Using Library Compiler Syntax”](#)
 - *Synopsys Logic Library Reference Manual*
-

Basic Syntax Rules

These are the Library Compiler syntax rules:

- Names are case-sensitive but not limited in length.
 - Each identifier must be unique within its scope.
 - Statements can span multiple lines, but each line before the last must end with a continuation character (\).
-

Group Statements

A group object is a named collection of statements that defines a library, a cell, a pin, a timing arc, a bus, and so forth.

This is the syntax of a group statement:

```
group_name (name) {  
    ... statements ...  
}
```

Attribute Statements

An attribute statement defines the characteristics of a specific object. Attributes are defined within a group and can be either simple or complex. The two types of attributes are distinguished by their syntax. All simple attributes use the same general syntax. Complex attributes, however, have different syntactic requirements.

This is the syntax of a simple attribute:

```
attribute_name : attribute_value
```

This is the syntax of a complex attribute:

```
attribute_name (parameter1 [, parameter2, parameter3 ...])
```

Task 2: Creating a Library

The examples in this chapter describe the development of a technology library.

References

For more information about creating a library, see

- [Chapter 4, “Using Library Compiler Syntax”](#)
 - *Synopsys Logic Library Reference Manual*
-

Procedure for Creating a Library

These are the steps for creating a library:

1. Determine the technology.
 2. Select a delay model.
 3. Determine where the library is used (that is, synthesis or simulation).
 4. Describe the library in Library Compiler syntax.
-

Determining the Technology

The technology is CMOS (default).

Selecting a Delay Model

Select the model to be used in delay calculations. The supported delay model is
table_lookup

This model is the CMOS nonlinear delay model.

Determining Where This Library Is Used

Determine whether this library is used for synthesis or for simulation. This chapter describes a synthesis library.

Describing the Library

Complete the following tasks to describe the library in the Library Compiler syntax:

- Name the library.
- Define the library group.
- Specify the library characteristics, such as the technology and delay model.

Naming the Library

Use these extensions for library file names:

.lib

Technology library source files

.db

Compiled technology libraries in Synopsys database format

Defining the library Group

The `library` group statement is the first executable line in your library source file. It is good to use the library name provided by the vendor. However, you can also select the name.

Example

This `library` group statement names a library called `my_lib`:

```
library(my_lib) {  
    ...  
}
```

Specifying library Group Attributes

At the library level, simple attributes define features of the library, such as units of time, voltage, current, delay model, bus naming style, and documentation.

A complex attribute defines the technology, such as voltage map and unit for capacitive load.

technology Attribute

The `technology` attribute in the `library` group defines the technology used for the library. Valid value is `cmos`.

delay_model Attribute

The `delay_model` attribute defines the delay model used in the library. Valid value is `table_lookup`, which specifies the nonlinear delay model.

Example

In this example, the `library` group specifies CMOS technology and the nonlinear delay model:

```
library(NLDM) {  
    technology(cmos);  
    delay_model : table_lookup;  
}
```

Task 3: Describing the Environment

Variations in operating temperature, supply voltage, and manufacturing process cause performance variations in electronic networks. Use the Library Compiler environment attributes to model these variations. Downstream tools use these environment attributes to modify the synthesis and optimization environment.

Using different operating conditions and k-factors, you can evaluate the timing of a circuit under different environmental conditions. Normally, the delay values specified for the cells in a technology library specify the set for the nominal operating conditions.

References

For more information about describing the environment, see [Chapter 6, “Building Environments](#).

Procedure

The following steps describe the environment:

1. Model delay calculation scaling factors.
2. Define default attributes.
3. Define operating conditions.
4. Model the wire load.

Modeling Delay Calculation Scaling Factors

The Design Compiler tool calculates delay estimates by using the scaling factors set in the technology library environment. These k-factors (attributes that begin with k_) are multipliers that scale cell delays as a function of process, temperature, and voltage. Every part of a cell description that is part of the delay equation is individually scaled with its own k-factors. The k-factors are used with operating conditions groups.

The scaling factors for your library depend on the timing delay model you use. For information about the delay analysis equations used by the downstream tools, see [Chapter 10, “Nonlinear Delay Model”](#) and [Chapter 11, “Timing Arcs”](#).

Example

```
library(example) {  
    k_process_cell_fall : 1.0;  
    k_process_cell_rise : 1.0;  
}
```

For a description of delay scaling attributes, see [Chapter 6, “Building Environments”](#).

Defining Default Attributes

All environment attributes have built-in default settings for typical cases. If you run it without variables, the Design Compiler tool uses these typical cases during optimization. As an alternative, you can create your own default settings.

You can set global default attributes at the library level for pins, timing, wire load, chip utilization, operating conditions group, and cell leakage power. For a description of library-level default attributes, see [Chapter 6, “Building Environments”](#).

Example

```
library(example) {  
    default_max_transition : 1.0;  
    default_cell_leakage_power : 1.0;  
}
```

Defining Operating Conditions

A set of operating condition definitions specifies the temperature, voltage, process, and RC tree model to use when analyzing the timing and other characteristics of a design. Operating condition sets are useful for testing your design in predefined, simulated environments. A library can contain multiple operating conditions.

Example

Define the operating conditions in an `operating_conditions` group within the `library` group.

```
library (example) {
    operating_conditions (WCCOM) {
        process : 1.5 ;
        temperature : 70 ;
        voltage : 4.75 ;
        tree_type : worst_case_tree ;
    }
}
```

`name`

Name of the `operating_conditions` group to identify the set of operating conditions.

`process`

A factor to account for variations in the outcome of the actual semiconductor manufacturing steps. Typically, 1.0 for normal operating conditions.

`temperature`

The ambient temperature in which the design is to operate. The value is a floating-point number.

`voltage`

The operating voltage of the design.

`tree_type`

The definition for the environment interconnect model. The Design Compiler tool uses the interconnect model to select the formula for calculating interconnect delays.

During optimization, the Design Compiler tool selects the operating conditions in this order of precedence:

1. The `operating_conditions` group specified by the `dc_shell` `set_operating_conditions` command
2. The `operating_conditions` group defined by the Library Compiler `default_operating_conditions` attribute
3. Nominal operating conditions defined in the `library` group

Modeling the Wire Load

The Design Compiler tool estimates interconnect wiring delays using the `wire_load` and `wire_load_selection` groups. These groups define the estimated wire length as a function of fanout and the scaling factors when determining wire resistance, capacitance, and area for a given length of wire.

Specifying Values for the 3-D Wire Delay Lookup Table

You can assign the following values to `variable_1`, `variable_2`, and `variable_3` in the lookup table templates for wire delays specified by the `lu_table_template` group.

```
fanout_number | fanout_pin_capacitance | driver_slew;
```

The values that you can assign to the variables of a table specifying wire delay depend on whether the table is one-, two-, or three-dimensional.

Example

```
lu_table_template(wire_delay_table_template) {
    variable_1 : fanout_number;
    variable_2 : fanout_pin_capacitance;
    variable_3 : driver_slew;
    index_1 ("1.0 , 3.0");
    index_2 ("0.12, 4.24");
    index_3 ("0.1, 2.7, 3.12");
}
lu_table_template(trans_template) {
    variable_1 : total_output_net_capacitance;
    index_1 ("0.0, 1.5, 2.0, 2.5");
}
wire_load("05x05") {
    resistance : 0 ;
    capacitance : 1 ;
    area : 0 ;
    slope : 0.186 ;
    fanout_length(1,0.39) ;
    interconnect_delay(wire_delay_table_template)
        values("0.00,0.21,0.3", "0.11,0.23,0.41", \
        "0.00,0.44,0.57", "0.10 0.3, 0.41");
}
```

name

The name ("05x05" in the example) identifies the `wire_load` group.

resistance

The wire resistance per unit length of interconnect wire.

capacitance

The capacitance per unit length of interconnect wire.

area

The area per unit length of interconnect wire.

slope

Characterizes linear fanout length behavior beyond the scope of the longest length described by the `fanout_length` attribute. This is in units of area per fanout.

fanout_length

The values that represent fanout and length, respectively. The first is an integer specifying the total number of pins minus the one driven by the output. The second is a statistical value that represents the amount of metal in a network with a given number of pins.

interconnect_delay

An optional complex attribute that specifies the lookup table template and the wire delay values.

You can define multiple `wire_load` groups in a technology library, but all the `wire_load` and `operating_conditions` groups must have unique names.

If you define a `wire_load_selection` group in the `library` group, the Design Compiler tool automatically selects a `wire_load` group for wire load estimation, based on the total cell area of the design.

For more information about defining wire loads and automatically selecting `wire_load` groups, see [Chapter 6, "Building Environments"](#).

Task 4: Defining the Cells

Cell descriptions provide information about the area, function, timing, and power of each component in an ASIC technology library. Specifically, a cell description includes

Structure

The cell, bus, and pin structure that describes the connection of each cell to the outside world.

Function

The logical function of each output pin of the cell used by the Design Compiler tool to map the logic of a design to the ASIC technology library.

Timing

Timing analysis and design optimization information, such as the parameters for pin-to-pin timing relationships, delay calculations, and timing constraints for sequential cells.

Power

State-dependent and path-dependent power modeling information.

For more information, see [Chapter 12, “Modeling Power and Electromigration”](#).

Other synthesis parameters

Parameters that describe the area and design rules. For example, each cell can have a definition of the maximum fanout of an output pin.

Note:

Technology libraries are processed without regard to explicitly specified units of measure for delay, capacitance, resistance, voltage, temperature, or area. Choose consistent units (nanoseconds, picofarads, kilohms, volts, and degrees celsius) throughout the library. The units used to describe area depend on the ASIC technology. For gate arrays, the value of area for a given cell is the number of gates. Standard cells and cell-based custom designs use a measure of surface area, such as square micrometers. Pads are specified to have zero area because they do not occupy area in the logic core of the chip.

References

See these sources for more information about defining cells:

- [Chapter 4, “Using Library Compiler Syntax”](#)
- [Chapter 5, “Building a Technology Library”](#)
- [Chapter 8, “Defining Sequential Cells”](#)
- [Chapter 21, “Defining I/O Pads”](#)
- *Synopsys Logic Library Reference Manual*

Procedure

Cells are mainly of two types: core cells and input or output pads. To define cells:

1. Define core cells
2. Define input or output pads

Defining Core Cells

For the Design Compiler tool to map the technology and optimize the design descriptions, the technology library must contain a minimum set of cells. The Design Compiler tool issues a warning message if the design contains a cell that is not included in the library or libraries associated with the design.

The minimum cell set for a CMOS technology library is:

- One of the following:
 - 2-input AND gate and 2-input OR gate
 - 2-input NOR gate
- Inverter
- Three-state buffer
- D flip-flop with preset, clear, and complementary output values
- D latch with preset, clear, and complementary output values

The example libraries contain these cells:

- 2-input AND gate
- 2-input OR gate
- Inverter
- Three-state buffer
- D flip-flop

To define core cells,

1. See the datasheet
2. Create the cell and specify a name
3. List the pins, specifying the direction and capacitive load for each pin
4. Determine pin functions (combinational or sequential logic)
5. Describe timing in the cell

Referring to the Datasheet

Datasheets contain information about area, capacitance, intrinsic delays, and extrinsic delays based on loading.

The unit of measure for capacitance does not matter to the Design Compiler tool. However, ensure that the units are consistent across the library and with the timing equation factors used to describe the library. The units of capacitance are usually standard load units or picofarads.

The cell rise and fall of a component are the rise and fall delays through a macro cell, regardless of delays caused by layout, loading, and environment.

Rise transition and fall transition are factors used to calculate the extrinsic delay based on the capacitive load of the output pin.

[Figure 2-1](#) shows a datasheet example for a 2-input AND gate. The information from this datasheet is described in a library that uses the `table_lookup` delay model.

You can transcribe the information from the datasheet into the cell descriptions in the Library Compiler syntax, as shown in the following sections.

Figure 2-1 Datasheet Example for a 2-input AND Gate

Name of standard driver

Function of macro cell

Number of gates and I/O used by cell

CELL NAME	FUNCTION	CELL COUNT	
		GATE	I/O
AN2	2-INPUT AND	2	0
AN2P		2	0

Name of high driver

LOGIC SYMBOL

TRUTH TABLE ← Shows the functionality

cell_rise

Delay Path

rise_transition

cell_fall

fall_transition

AC CHARACTERISTICS

CELL	PATH	TpLH		TpHL	
		Tup	Kup	Tdn	Kdn
AN2	A,B->Z	0.39	0.122	0.53	0.038
AN2P	A,B->Z	0.51	0.053	0.58	0.023
UNIT		nS	nS/LU	nS	nS/LU

Load capacitance of input. It is also a factor in delay calculation.

related_pin

Load drive capability of output

INPUT LOAD (LU)

OUTPUT DRIVE (LU)

CELL	PIN	A,B
AN2		1.0
AN2P		1.0

capacitance

CELL	PIN	Z
AN2		25
AN2P		57

max_fanout

Specifying the Cell Name

Specify the cell name using a `cell` group. The `cell` group defines a single cell in a technology library.

Example

This statement for the 2-input AND cell names the cell and defines its area.

```
cell(AN2) { /* 2-input AND gate */
    area : 2 ;
    ...
}
area
```

Defines the cell area. The default is zero. Specify the `area` attribute in the library to use the library for area optimization. The exception is pad cells that have zero area and do not occupy area in the logic core of the chip.

Listing the Pins

To list the pins of a cell, define the pins using the `pin` groups within a `cell` group.

Example

In the 2-input AND cell, two pin statements list three pins: A, B, and Z.

```
pin(A,B) {
    direction : input ;
    capacitance : 1 ;
}
pin(Z) {
    direction : output ;
    function : "A * B" ;
    timing() {
        cell_rise(scalar) {values( " 0.39 ")} ;
        cell_fall(scalar) {values( " 0.53 ")} ;
        rise_transition(scalar) {values( " 0.122 ")} ;
        fall_transition(scalar) {values( " 0.038 ")} ;
        related_pin : "A B" ;
    }
}
```

The `pin` group example demonstrates pin direction, capacitance, function, and timing specifications. You define these specifications by using the following attributes:

`direction`

Defines the direction of each pin. In the example, A and B are defined as input pins and Z as an output pin.

capacitance

Defines the input pin load (input capacitance) on the network.

function

Defines the logic function of an output pin in terms of the cell's input or inout pins. In the example, the function of pin Z is defined as the logical AND of pins A and B.

timing

Describes timing groups. A timing group describes the following:

- A pin-to-pin delay
- A timing constraint such as setup and hold

In the example, the timing group for pin Z describes the delays between pin Z and pins A and B.

See “[Describing Timing](#)” on page 2-18 for more information about timing.

Determining Pin Functions

To describe the pin functions, use the function attribute within the pin group.

For combinational cells, the function attribute specifies the Boolean function of an output pin in terms of the cell's input and inout pins.

To define sequential cell behavior, use the following groups:

- ff – For a flip-flop.
- latch – For a latch.
- statetable – For general sequential behavior.

Both the ff and latch groups have two state-variables that represent the noninverted and inverted outputs. Use these state variables in the function attribute to define the output pin function.

The statetable group defines a state table. The Library Compiler tool creates an internal node that represents the output function of the state table. Specify this internal node in an internal_node or state_function attribute when you define the output pin's function.

[Example 2-1](#) uses an internal_node attribute statement for the function of the output pins.

Example 2-1 D Flip-Flop With statetable Group

```
cell(flipflop1) {  
    area : 7 ;  
    pin(D) {  
        direction : input ;
```

```

capacitance : 1.3 ;
timing() {
    timing_type : setup_rising ;
    cell_rise(scalar) {values( " 0.9 ")} ;
    cell_fall(scalar) {values( " 0.9 ")} ;
    related_pin : "CP" ;
}
timing() {
    timing_type : hold_rising ;
    cell_rise(scalar) {values( " 0.5 ")} ;
    cell_fall(scalar) {values( " 0.5 ")} ;
    related_pin : "CP" ;
}
pin(CP) {
    direction : input ;
    capacitance : 1.3 ;
    min_pulse_width_high : 1.5 ;
    min_pulse_width_low : 1.5 ;
}
statetable (" D      CP", "IQ      IQN") {
    table :   " L/H   R : - - : L/H   H/L,\n              -     ~R : - - : N       N" ;
}
pin(Q) {
    direction : output ;
    internal_node : "IQ" ;
    timing() {
        timing_type : rising_edge ;
        cell_rise(scalar) {values( " 1.11 ")} ;
        cell_fall(scalar) {values( " 1.43 ")} ;
        rise_transition(scalar) {values( " 0.1513 ")} ;
        fall_transition(scalar) {values( " 0.0544 ")} ;
        related_pin : "CP" ;
    }
}
pin(QN) {
    direction : output ;
    internal_node : "IQN"
    timing() {
        timing_type : rising_edge ;
        cell_rise(scalar) {values( " 1.58 ")} ;
        cell_fall(scalar) {values( " 1.56 ")} ;
        rise_transition(scalar) {values( " 0.1513 ")} ;
        fall_transition(scalar) {values( " 0.0544 ")} ;
        related_pin : "CP" ;
    }
}
}

```

All the definitions in [Example 2-2](#) are same as in [Example 2-1](#), except that

- The `ff` group statement replaces the `statetable` group statement
- The `function` attribute, rather than the `internal_node` attribute, defines the output pin's function

The D flip-flop defines two variables, IQ and IQN. The next_state equation determines the value of IQ after the next_clocked_on transition. In [Example 2-2](#), IQ is assigned the value of the D input.

Example 2-2 D Flip-Flop With ff Group

```
cell(flipflop1) {
    area : 7 ;
    pin(D) {
        direction : input;
        ...
    }
    pin(CP) {
        direction : input;
        ...
    }
    ff(IQ,IQN) {
        next_state : "D" ;
        clocked_on : "CP" ;
    }
    pin(Q) {
        direction : output ;
        function : "IQ" ;
        ...
    }
    pin(QN) {
        direction : output ;
        function : "IQN";
        ...
    }
}
```

Describing Timing

Timing is divided into two major areas: describing delay (the actual circuit timing) and specifying constraints (boundaries). Timing arcs are paths followed by path tracers during path analysis, with netlist interconnect information. Timing arcs can be delay arcs or constraint arcs. Each timing arc has a startpoint and an endpoint. The startpoint can be an input, output, or inout pin. The endpoint is always an output or inout pin. The only exception is a constraint timing arc, such as a setup or hold constraint between two input pins.

Describe timing delay and constraints in the `timing` group in a `bundle`, `bus`, or `pin` group within a cell. Timing groups contain the information that the Design Compiler tool needs to

model timing arcs and trace paths. The `timing` groups define the timing arcs through a cell and the relationships between clock and data input signals.

Optionally, the timing arc or arcs in a design can be identified with a name or names entered with the `timing` group attribute using the following syntax:

```
timing (name | name_list);
```

For naming details, see the section on using the `timing` group in [Chapter 11, “Timing Arcs”](#).

The following attribute is required in all `timing` groups:

`related_pin`

This attribute defines the pin or pins representing the startpoint of a timing arc.

You can define more than one `timing` group for each pin. Define two or more `timing` groups to allow multiple timing arcs for each path between an input and output pin.

You can also use the `timing` group attribute to define multiple timing arcs, as in a case where a timing arc has multiple related pins or when the timing arc belongs to a `timing` group within a `bus` or `bundle` group.

All delay information in a library refers to an input-to-output pin pair or an output-to-output pin pair. You can define these types of delay:

`cell delay`

The fixed delay from input to output pins.

`transition delay`

The time it takes the driving pin to change state. Transition delay attributes represent the resistance encountered in making logic transitions.

`slope sensitivity`

The incremental time delay due to slow change of input signals.

[Table 2-1](#) shows the groups you can specify to describe the following types of delay in the CMOS nonlinear delay model.

Table 2-1 Delay Types and Valid Attributes

Delay type	Valid attributes
Cell	<code>cell_rise</code> , <code>cell_fall</code>
Transition	<code>rise_transition</code> , <code>fall_transition</code>
Constraint	<code>rise_constraint</code> , <code>fall_constraint</code>

You can set various timing constraints, such as these:

setup and hold arcs

Set these constraints to ensure that a data signal has stabilized, before latching its value.

recovery and removal arcs

Use the recovery timing arc and the removal timing arc for asynchronous control pins such as clear and preset.

You can also set state-dependent and conditional constraints.

For setting constraints and defining delay with different delay models, see [Chapter 11, “Timing Arcs”](#). The “Generating Library Reports” chapter of the *Library Quality Assurance System User Guide* describes how to get information about timing in cells by using the `report_lib -timing` command.

Defining I/O Pads

I/O pads are the special cells at the chip boundaries that allow communication with the world outside the chip. Their characteristics distinguish them from the other cells that make up the core of an integrated circuit. Some of these distinguishing characteristics are

- Longer delays
- Greater drive capabilities
- Voltage levels for transferring logic signals to the core or logic values from the core

You must capture these characteristics in the library so that IC designers can insert the correct pads during synthesis.

To define I/O pads, you combine the `library`, `cell`, and `pin` group attributes that describe input, output, and bidirectional pad cells.

Note:

I/O pads are often defined in a separate library.

Use these steps to define I/O pads:

1. Identify pad cells
2. Describe multicell pads
3. Define units for pad cells
4. Identify area resources
5. Describe input pads

6. Describe output pads

7. Model wire load

Identifying Pad Cells

When it synthesizes a chip, the Design Compiler tool uses the methods that you used to identify cells as I/O pads. The Design Compiler tool filters pad cells out of normal core optimization and treats them differently during technology translation.

These attributes identify pads:

`pad_cell` and `auxiliary_pad_cell`

Attributes that identify a cell as an I/O pad.

`pad_type : clock`

Attribute that identifies a clock driver pad cell.

`is_pad`

Attribute on pins that identifies logical I/O pad pins.

`direction`

Attribute defining whether a pad is an input, output, or bidirectional pad. For bidirectional pins, you can define multiple `driver_type` attributes for the same pin to model both output and input behavior.

Describing Multicell Pads

Pads can have multiple cells—for example, a pad cell and a driver cell.

You can describe I/O pads in one of two ways:

- As one cell with many attributes
- As a collection of cells, each with attributes affecting the operation of the logical pad

A library using the first method to describe multiple cells contains a different cell for each possible combination of pad characteristics. A library using the second method contains a smaller number of components, but each type of pad must be individually constructed from specific components.

These are attributes that describe multiple cells:

`auxiliary_pad_cell`

Indicates that the cell can be used as part of a logical pad.

multicell_pad_pin

Identifies the pins to connect to create a working multicell pad or an auxiliary pad cell.

connection_class

Identifies the pins that are to be connected to pins on other cells.

Defining Units for Pad Cells

To handle pads for full-chip synthesis, the Design Compiler tool needs the physical quantities of time, capacitance, resistance, voltage, current, and power. Use these attributes to supply this information:

capacitive_load_unit

Defines the capacitance associated with a standard load.

pulling_resistance_unit

Defines the unit for the `pulling_resistance` attribute for pull-up and pull-down devices on pads.

voltage_unit

Defines the unit for the `input_voltage` and `output_voltage` groups that define input or output voltage ranges for cells.

current_unit

Defines current values for the `drive_current` and `pulling_current` attributes.

Identifying Area Resources

For full-chip synthesis, the Design Compiler tool needs the `area` attribute, which describes the area resources associated with the chip.

area

Describes the consumption of core and area resources by cells and wiring.

Describing Input Pads

To describe the input voltage characteristics of pads with hysteresis, use these components:

input_voltage group

Defines input voltage ranges that can be assigned to pad input pins.

hysteresis attribute

Identifies whether a pad has hysteresis. Pads with hysteresis sometimes have derating factors that are different from the factors for the cells in the core.

Note:

You can also use the `input_voltage` group to specify voltages ranges for standard cells.

Describing Output Pads

Output pad descriptions include the output voltage characteristics and the drive-current rating of output and bidirectional pads. Additionally, an output pad description includes information about the slew rate of the pad.

Use the following to describe output pads:

output_voltage group

Defines output voltage ranges for the output pin of a pad cell.

drive_current attribute

Defines the drive current supplied by the pad buffer in units consistent with the `current_unit` attribute.

slew_control attribute

Qualitatively measures the level of slew-rate control associated with an output pad. Additionally, there are eight attributes that quantitatively measure the behavior of the pad on rising and falling transitions. See “[Slew-Rate Control](#)” on page 21-9 for information about these attributes.

Note:

You can also use the `output_voltage` group to specify voltages ranges for standard cells.

Modeling Wire Load

You can define several `wire_load` groups to contain all the information the Design Compiler tool needs to estimate interconnect wiring delays. Estimated wire loads for pads can be significantly different from those of core cells.

The wire load model for the net connecting an I/O pad to the core needs to be handled separately, because it is usually longer than most other nets in the circuit. (Some I/O pad nets extend completely across the chip.)

See “[Modeling the Wire Load](#)” on page 2-9 for an example of the `wire_load` group.

Task 5: Describing Application-Specific Data

There are other attributes and groups that you can specify in your technology library for fixing design rules or modeling the design for testing or power.

References

For information about describing application-specific data, see the following chapters:

- [Chapter 6, “Building Environments”](#)
 - [Chapter 7, “Defining Core Cells”](#)
 - [Chapter 9, “Defining Test Cells”](#)
 - [Chapter 12, “Modeling Power and Electromigration”](#)
-

Procedure for Describing Application-Specific Data

Use this procedure to describe your data. Each procedural step is described in detail in the following sections.

1. Set the design rules
 2. Model for testable circuits
 3. Model for power
 4. Add synthesis-to-layout features
-

Setting Design Rules

In this task, you set the design rule constraints on cells; that is, you specify values for fanout and capacitive load.

Some designs have input load limitations that an output pin can drive, regardless of any loading contributed by interconnect metal layers. To limit the number of inputs on the same net driven by an output pin, you can

- Define a maximum fanout value on a cell’s output pins and a fanout load on its input pins
- Define a maximum transition time on an output pin
- Limit the total capacitive load that an output pin can drive

To specify load values, use the following attributes:

fanout_load

Specifies how much to add to the fanout on the net.

max_fanout

Specifies the maximum number of loads a pin can drive.

max_transition

Specifies the maximum limits of the transition delay on a network (total capacitive load on a network).

max_capacitance

Specifies the maximum total capacitive load that an output pin can drive.

min_fanout

Specifies the minimum number of loads that a pin can drive.

min_capacitance

Specifies the minimum total capacitive load that an output pin can drive.

See “[Describing Design Rule Checks](#)” on page 7-35 for more information about the design rule attributes.

Modeling for Testable Circuits

The DFT Compiler tool facilitates the design of testable circuits with minimal speed and area overheads, and generates an associated set of test vectors automatically. The DFT Compiler tool uses either full-scan or partial-scan methodology to add scan cells to your design and help make a design controllable and observable.

To use the DFT Compiler tool, add test-specific details of scannable cells to your technology libraries. Identify scannable flip-flops and latches and select the types of cells that are not scannable which they replace for a given scan methodology. See [Chapter 9, “Defining Test Cells,”](#) for information about scan cells.

Modeling for Power

You can model static and dynamic power for CMOS technology. The three components of power dissipation are leakage power, short-circuit (or internal) power, and switching power.

Leakage Power

Leakage power is the static (or quiescent) power dissipated when a gate is not switching. It is important to model leakage power for designs that are in an idle state most of the time.

Represent leakage power information with the cell-level `cell_leakage_power` attribute and associated library-level attributes that specify scaling factors, units, and a default.

The following example describes leakage power.

```
library(power_example) {
    leakage_power_unit : 1nW ;
    default_cell_leakage_power : 0.1 ;
    ...
    cell(AN2) {
        ...
        cell_leakage_power : 0.2 ;
        leakage_power() {
            when : "!A" ;
            values : (2.0) ;
        }
    }
}
```

Short-Circuit Power

Short-circuit or internal power is the power dissipated whenever a pin makes a transition.

Library developers can choose one of the following options:

- Include the effect of the output capacitance in the `internal_power` group (defined in a `pin` group within a `cell` group), which gives the output pins zero capacitance
- Give the output pins a real capacitance, which causes them to be included in the switching power, and model only the short-circuit power as the cell's internal power (in the `internal_power` group)

Switching Power

Switching (or interconnect) power is the power dissipated by the capacitive load on a net whenever the net makes a logical transition. Power is dissipated when the capacitive load is charged or discharged. Switching power (along with internal power) is used to compute the design's total dynamic power dissipation.

For information about the attributes used in modeling static and dynamic power, see [Chapter 6, “Building Environments”](#) and [Chapter 12, “Modeling Power and Electromigration”](#).

Technology Library With Delay Example

The following example shows a source file for a technology library that uses the nonlinear delay model.

```
library(NLDM) {
  technology (cmos) ; /* technology */
  delay_model : table_lookup; /* delay model */

  default_inout_pin_cap : 1.0; /* default attributes */
  default_input_pin_cap : 1.0;
  default_output_pin_cap : 0.0;
  default_fanout_load : 1.0;

  k_process_pin_cap : 0.0; /* delay calculation
    scaling factors */
  k_process_wire_cap : 0.0;
  k_temp_pin_cap : 0.0;
  k_temp_wire_cap : 0.0;
  k_volt_pin_cap : 0.0;
  k_volt_wire_cap : 0.0;

  slew_lower_threshold_pct_fall : 30.0; /* threshold definitions */
  slew_upper_threshold_pct_fall : 70.0;
  slew_lower_threshold_pct_rise : 30.0;
  slew_upper_threshold_pct_rise : 70.0;
  input_threshold_pct_fall : 50.0;
  input_threshold_pct_rise : 50.0;
  output_threshold_pct_fall : 50.0;
  output_threshold_pct_rise : 50.0;

  time_unit : "1ns"; /* library units */
  pulling_resistance_unit : "100ohm";
  voltage_unit : "1V";
  current_unit : "1mA";
  capacitive_load_unit(1,pf);
  nom_process : 1.0;
  nom_temperature : 25.0;
  nom_voltage : 5.0;

  operating_conditions(WCCOM) {
    /* operating condition models */
    process : 1.5 ;
    temperature : 70 ;
    voltage : 4.75 ;
    tree_type : "worst_case_tree" ;
  }
  operating_conditions(WCIND) {
    process : 1.5 ;
    temperature : 85 ;
    voltage : 4.75 ;
    tree_type : "worst_case_tree" ;
  }
}
```

```
}

operating_conditions(WCMIL) {
    process : 1.5 ;
    temperature : 125 ;
    voltage : 4.5 ;
    tree_type : "worst_case_tree" ;
}
operating_conditions(BCCOM) {
    process : 0.6 ;
    temperature : 0 ;
    voltage : 5.25 ;
    tree_type : "best_case_tree" ;
}
operating_conditions(BCIND) {
    process : 0.6 ;
    temperature : -40 ;
    voltage : 5.25 ;
    tree_type : "best_case_tree" ;
}
operating_conditions(BCMIL) {
    process : 0.6 ;
    temperature : -55 ;
    voltage : 5.5 ;
    tree_type : "best_case_tree" ;
}
wire_load("05x05") { /* wire load models */
    resistance : 0 ;
    capacitance : 1 ;
    area : 0 ;
    slope : 0.186 ;
    fanout_length(1,0.39) ;
}
wire_load("10x10") {
    resistance : 0 ;
    capacitance : 1 ;
    area : 0 ;
    slope : 0.311 ;
    fanout_length(1,0.53) ;
}
wire_load("20x20") {
    resistance : 0 ;
    capacitance : 1 ;
    area : 0 ;
    slope : 0.547 ;
    fanout_length(1,0.86) ;
}
wire_load("30x30") {
    resistance : 0 ;
    capacitance : 1 ;
    area : 0 ;
    slope : 0.782 ;
    fanout_length(1,1.40) ;
}
```

```
wire_load("40x40") {
    resistance : 0 ;
    capacitance : 1 ;
    area : 0 ;
    slope : 1.007 ;
    fanout_length(1,1.90) ;
}
wire_load("50x50") {
    resistance : 0 ;
    capacitance : 1 ;
    area : 0 ;
    slope : 1.218 ;
    fanout_length(1,1.80) ;
}
wire_load("60x60") {
    resistance : 0 ;
    capacitance : 1 ;
    area : 0 ;
    slope : 1.391 ;
    fanout_length(1,1.70) ;
}
wire_load("70x70") {
    resistance : 0 ;
    capacitance : 1 ;
    area : 0 ;
    slope : 1.517 ;
    fanout_length(1,1.80) ;
}
wire_load("80x80") {
    resistance : 0 ;
    capacitance : 1 ;
    area : 0 ;
    slope : 1.590 ;
    fanout_length(1,1.80) ;
}
wire_load("90x90") {
    resistance : 0 ;
    capacitance : 1 ;
    area : 0 ;
    slope : 1.64 ;
    fanout_length(1,1.9) ;
}
/* Define one dimensional lookup table of size 4 */
lu_table_template(trans_template) {
    variable_1 : input_net_transition;
    index_1 ("0.0, 0.5, 1.5, 2.0");
}
/* Define template of size 3x3 */
lu_table_template(constraint_template) {
    variable_1 : constrained_pin_transition;
    variable_2 : related_pin_transition;
    index_1 ("0.0, 0.5, 1.5");
    index_2 ("0.0, 2.0, 4.0");
```

```
}

cell(AN2) {                                /* 2-input AND gate */
    area : 2 ;
    pin(A,B) {
        direction : input ;
        capacitance : 1 ;
    }
    pin(Z) {
        direction : output ;
        function : "A * B" ;
        timing() {
            rise_propagation(scalar) {
                values ("0.12") ;
            }
            fall_propagation(scalar) {
                values ("0.12") ;
            }
            rise_transition(trans_template) {
                values ("0.1, 0.15, 0.20, 0.29") ;
            }
            fall_transition (trans_template) {
                values ("0.1, 0.15, 0.20, 0.29") ;
            }
            related_pin : "A B" ;
        }
    }
}
cell(OR2) {                                /* 2-input OR gate */
    area : 2 ;
    pin(A B) {
        direction : input ;
        capacitance : 1 ;
    }
    pin(Z) {
        direction : output ;
        function : "A + B" ;
        timing() {
            rise_propagation(scalar) {
                values ("0.12") ;
            }
            fall_propagation(scalar) {
                values ("0.12") ;
            }
            rise_transition(trans_template) {
                values ("0.1, 0.15, 0.20, 0.29") ;
            }
            fall_transition (trans_template) {
                values ("0.1, 0.15, 0.20, 0.29") ;
            }
            related_pin : "A B" ;
        }
    }
}
```

```
cell(TRI_INV2) {                                /* three-state INVERTER */
    area : 3 ;
    pin(A) {
        direction : input ;
        capacitance : 2 ;
    }
    pin(E) {
        direction : input ;
        capacitance : 2 ;
    }
    pin(Z) {
        direction : output ;
        function : "A'" ;
        three_state : "E'" ;
        timing() {
            rise_propagation(scalar) {
                values ("0.12") ;
            }
            fall_propagation(scalar) {
                values ("0.12") ;
            }
            rise_transition(trans_template) {
                values ("0.1, 0.15, 0.20, 0.29") ;
            }
            fall_transition (trans_template) {
                values ("0.1, 0.15, 0.20, 0.29") ;
            }
            related_pin : "A" ;
        }
        timing() {
            timing_type : three_state_enable;
            rise_propagation(scalar) {
                values ("0.12") ;
            }
            fall_propagation(scalar) {
                values ("0.12") ;
            }
            rise_transition(trans_template) {
                values ("0.1, 0.15, 0.20, 0.29") ;
            }
            fall_transition (trans_template) {
                values ("0.1, 0.15, 0.20, 0.29") ;
            }
            related_pin : "E" ;
        }
        timing() {
            timing_type : three_state_disable ;
            rise_propagation(scalar) {
                values ("0.12") ;
            }
            fall_propagation(scalar) {
                values ("0.12") ;
            }
        }
    }
}
```

```

        rise_transition(trans_template) {
            values ("0.1, 0.15, 0.20, 0.29");
        }
        fall_transition (trans_template) {
            values ("0.1, 0.15, 0.20, 0.29");
        }
        related_pin : "E" ;
    }
}
cell(FF1) {                                     /* D FLIP-FLOP */
    area : 7 ;
    pin(D) {
        direction : input ;
        capacitance : 1.3 ;
        timing() {
            timing_type : setup_rising ;
            related_pin : "CP" ;
            rise_constraint(constraint_template) {
                values ("0.0, 0.13, 0.19", \
                         "0.21, 0.23, 0.41", \
                         "0.33, 0.37, 0.50");
            }
            fall_constraint(constraint_template) {
                values ("0.0, 0.14, 0.20", \
                         "0.22, 0.24, 0.42", \
                         "0.34, 0.38, 0.51");
            }
        }
        timing() {
            timing_type : hold_rising ;
            related_pin : "CP" ;
            rise_constraint(constraint_template) {
                values ("0.0, 0.13, 0.19", \
                         "0.21, 0.23, 0.41", \
                         "0.33, 0.37, 0.50");
            }
            fall_constraint(constraint_template) {
                values ("0.0, 0.14, 0.20", \
                         "0.22, 0.24, 0.42", \
                         "0.34, 0.38, 0.51");
            }
        }
    }
}

pin(CP) {
    direction : input ;
    capacitance : 1.3 ;
    min_pulse_width_high : 1.5 ;
    min_pulse_width_low : 1.5 ;
}
ff ("IQ", "IQN") {
    next_state : "D" ;
}

```

```
        clocked_on : "CP" ;
    }
pin(Q) {
    direction : output ;
    function: "IQ" ;
    timing() {
        timing_type : rising_edge ;
        rise_propagation(scalar) {
            values ("0.12");
        }
        fall_propagation(scalar) {
            values ("0.12");
        }
        rise_transition(trans_template) {
            values ("0.1, 0.15, 0.20, 0.29");
        }
        fall_transition (trans_template) {
            values ("0.1, 0.15, 0.20, 0.29");
        }
        related_pin : "CP" ;
    }
}
pin(QN) {
    direction : output ;
    function : "IQN"
    timing() {
        timing_type : rising_edge ;
        rise_propagation(scalar) {
            values ("0.12");
        }
        fall_propagation(scalar) {
            values ("0.12");
        }
        rise_transition(trans_template) {
            values ("0.1, 0.15, 0.20, 0.29");
        }
        fall_transition (trans_template) {
            values ("0.1, 0.15, 0.20, 0.29");
        }
        related_pin : "CP" ;
    }
}
}
/* end of library */
```

Task 6: Compiling the Library

This section describes how to compile a library.

Procedure

Compile a library using the shell command interface (lc_shell).

Using the shell interface,

1. Read the library source file into the tool.

```
lc_shell> read_lib path/libraries/tutorial.lib
```

2. Save the library memory file to a disk file in either Synopsys internal database (.db) format.

```
lc_shell> write_lib tutorial -format db \
           -output path/libraries/example.db
```

References

For further information about compiling a library, see [Chapter 3, “Using the Library Compiler Shell.”](#)

Task 7: Managing the Library

This section describes how to manage a library.

References

For more information about managing a library, see [Chapter 3, “Using the Library Compiler Shell.”](#)

Procedure

You can manage the libraries by using the following Library Compiler shell commands:

read

Loads a previously saved .db library.

report_lib

Generates a report of the contents of either a source library (.lib) or a compiled library (.db).

remove_lib

Removes a library from memory.

3

Using the Library Compiler Shell

You can access the Library Compiler tool from the shell (command-line) interface. The shell interface lets you type commands, arguments, and options at a system prompt.

To use the shell interface, you need to know about the following concepts described in this chapter:

- [About the Shell Interface](#)
- [Describing Library Files and Memory Files](#)
- [lc_shell Command Syntax](#)
- [Reading In Libraries](#)
- [Writing Library Files](#)
- [Generating Hierarchical .db Designs](#)
- [Verifying Hierarchical .db Designs](#)
- [Loading Compiled Libraries](#)
- [Adding Library-Level Information to Existing Libraries](#)
- [Accessing Library Information](#)
- [Reporting Library Information](#)
- [Library Scaling](#)

- [Removing Libraries From Memory](#)
- [Using Shell Commands](#)
- [Using Variables](#)
- [Productions](#)

For additional information about the shell commands, see the Design Compiler reference manuals or the Synopsys man pages.

About the Shell Interface

Like other command-oriented interfaces, the Library Compiler shell interface has both commands (directives) and variables (symbols or named values).

You use `lc_shell` commands to perform specific actions on a library. For example, you can use the `read_lib` command to compile a library source file into the Library Compiler tool.

The `lc_shell` variables let you specify one or more values, such as the modeling option or the search path.

To use the Library Compiler shell interface, you must know how to perform these tasks, which are described in the following sections:

- Start the Library Compiler command-line interface (`lc_shell`)
 - Understand the command log
 - Use setup files
 - Use `lc_shell` commands
 - Interrupt an `lc_shell` command
 - Exit the command-line interface
-

Starting the Command Interface

If you are using the UNIX operating system, make sure the `lc_shell` program is installed so that you can access it from your UNIX prompt. In the following examples, the percent sign (%) represents the generic UNIX system prompt.

To start `lc_shell` at your UNIX prompt, enter this command:

```
% lc_shell
```

Note:

If you cannot start `lc_shell`, see the Installation Guide for your hardware platform and correctly install the tool.

The `lc_shell` invocation command has several options:

-f

Executes a script file and starts the `lc_shell` interface:

```
prompt> lc_shell -f script_file
```

This example redirects output generated by `lc_shell` to a file named `output_file`:

```
prompt> lc_shell -f common.script > output_file
```

-no_init

Prevents Library Compiler setup files (described later in this section) from being read.

-no_init -f command_log.file

Use this option only when you have a command log or other script file that you want to include to reproduce a previous Library Compiler session.

-x *command_string*

Executes the `lc_shell` statement in *command_string* before displaying the initial `lc_shell` prompt. You can enter multiple statements. Separate each statement with a semicolon. For example:

```
prompt> lc_shell -x "echo Hi; echo World"
```

Note:

The pipe character (|) has no meaning in `lc_shell`. Use a backslash (\) to escape double quotation marks when executing a UNIX command. For example, the following command requires backslash characters before the double quotation marks to prevent the tool from ending the command prematurely:

```
lc_shell> sh "grep \"textstring \" my_file"
```

Understanding the Command Log

The command log records all `lc_shell` commands processed in a Library Compiler session, including setup file commands and variable assignments.

After the session, use the command log as a record of library exploration. You can also edit a command log to produce a script for a particular session.

The name of the command log file is determined by the `command_log_file` variable. The default command log name is `./lc_shell_command.log`. An `lc_shell` session creates a file named `lc_shell_command.log` in your directory. It also overwrites any existing `lc_shell_command.log` currently in your directory.

Set the `command_log_file` variable in a setup file, because the variable value does not take effect if you change it interactively during the session. For more information, see [“Using Setup Files” on page 3-5](#).

Note:

If you experience problems while using the Library Compiler tool, save the command log file for reference when you contact Synopsys. Then move or rename the command log file to prevent it from being overwritten by the next `lc_shell` session.

Using Setup Files

The defaults for many Library Compiler variables are set in Synopsys setup, or initialization, files. A setup file contains scripts that are automatically processed by `lc_shell` during initialization, unless you use the `-no_init` option. The name of the setup file is `.synopsys_lc.setup`.

Note:

You cannot include UNIX environment variables (such as `$SYNOPSYS`) in the `.synopsys_lc.setup` file.

The setup files reside in one of three types of directories, which are read by the tool in the following order:

1. The Synopsys root directory
2. Your home directory
3. The directory where you start the Library Compiler tool (the current directory)

If the setup files share commands or variables, the values in the last setup file read override the values in previously read files.

Using `lc_shell` Commands to Compile a Source File

To compile a technology library source file named `my_library.lib` into a Synopsys database, follow these steps:

1. Start the `lc_shell` interface, as described earlier in this chapter.

```
prompt> lc_shell
Library Compiler (TM)
...
Initializing...
lc_shell>
```

2. Read in the library source file (translate it into a Synopsys database and store the database in computer memory).

```
lc_shell> read_lib my_library.lib
```

3. Write the library to a system file.

```
lc_shell> write_lib my_library -output my_lib.db
```

4. Request the library reports.

```
lc_shell> report_lib my_lib
```

5. Exit the Library Compiler tool.

```
lc_shell> exit  
Thank you...
```

Support for gzip Format

The Library Compiler tool supports the gzip file format (*.gz). If a file has a *.gz extension, such as *library_name.lib.gz*, the tool determines that the file is gzipped. Then, it automatically unzips the file (as shown in the following example) and reads the data.

```
lc_shell> read_lib library_file_name.gz
```

Note:

The tool can only read gzip compressed files. No other compression file format is supported.

Interrupting Commands

If you enter the wrong options for a command or enter the wrong command, you can interrupt command processing by entering the interrupt sequence, Ctrl+C.

The time it takes for the command to respond to an interruption depends on the size of the library and the command being interrupted.

If the tool processes a script file and you interrupt one of its commands, the script processing is interrupted. No further script commands are processed, and control returns to lc_shell.

If you enter Ctrl+C three times before a command responds to your interruption, lc_shell is interrupted and exits with the following message:

```
Information: Process terminated by interrupt.
```

Leaving the Command Interface

You can use either the `quit` or the `exit` command to leave the Library Compiler tool and return to the operating system:

```
lc_shell> exit
```

```
Thank you...  
prompt>
```

or

```
lc_shell> quit
```

```
Thank you...  
prompt>
```

Optionally, you can specify an exit code value when you exit the command interface. The exit code value is passed to the system call (that is, `exit` or `quit`). The status of `lc_shell` is the exit status of the operating system that performs `exit` or `quit`. The default is 0.

```
lc_shell> exit
exit_code_value
Thank you...
prompt>
```

or

```
lc_shell> quit
exit_code_value
Thank you...
prompt>
```

On UNIX systems, you can also exit `lc_shell` by typing the end-of-file (EOF) sequence, `Ctrl+D`:

```
lc_shell> Ctrl D
Thank you...
prompt>
```

Note:

The `lc_shell` program does not automatically save information from the session. If you have set the `command_log_file` variable, the command log is written to the `./lc_shell_command.log` file or to the file specified by the `command_log_file` variable.

Describing Library Files and Memory Files

ASIC libraries are described by library source files, which are text files. For example,

path/libraries/my_library.lib

When you read a library into the Library Compiler tool,

- The file is translated into Synopsys database (.db) format.
- The tool stores each library in a memory file in this format:

pathname/filename:library

pathname

The directory from which the source file is read

filename

The name of the default disk file name for the compiled library

library

The library name you have specified

The default disk file name for a library is the library name with the suffix .db. For example, in UNIX, the memory file name of a library, my_lib, is

```
/usr/local/libraries/my_lib.db:my_lib
```

Two differences exist between UNIX library source files and Library Compiler memory files:

- Library source files exist in your host computer's file system; Library Compiler memory files exist in the working memory of the Library Compiler tool.
- Library source file names are complete file names; Library Compiler memory file names are composed of the path name and a library name. You can specify a compiled library by either its full memory file name or its library name if the library name is unique in memory.

Using search_path and Path Names

The lc_shell uses the underlying operating system to locate files; therefore, you must observe the operating system file name and path name conventions.

Table 3-1 shows the UNIX path specifications:

Table 3-1 Path Types for UNIX

Path types	UNIX
Absolute path	/user/libraries/my_library.db
Path relative to current directory	./my_library.db or my_library.db
Path relative to parent directory	../libraries/my_library.db
Path relative to a home directory	~designers/my_library.db

Use the `which` command to see where a file with a relative name can be found in the search path directories. For example, in lc_shell, enter

```
lc_shell> which my_library.db
{/usr/libraries/example/my_library.db}
```

lc_shell Command Syntax

An lc_shell command has the following components:

command name

The name of a command—for example, `exit`.

options

Option names are prefixed by a hyphen (-). A command can have zero or more options. For example, the `read_lib` command has multiple options.

You can abbreviate (truncate) option names, if the abbreviation is unique to the related command.

arguments

Command-specific values. Both commands and options can have arguments. For example, the `read_lib` command has an argument, such as the file name. The `write_lib` command has an option, `-output`. The `-output` option requires an argument, namely, the file name.

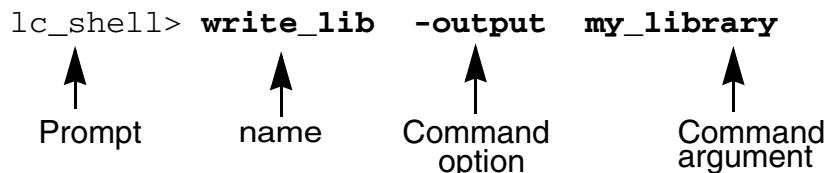
Note:

The lc_shell interface is not case-sensitive: alias and ALIAS are equivalent commands. Case is important, however, in file names, path names, and variables.

For example, `/usr/designers` is not the same path name as `/USR/DESIGNERS`. Also, the following two commands are not equivalent: `read_lib LIB1` and `read_lib lib1`.

Figure 3-1 shows a command with its name, options, and arguments labeled.

Figure 3-1 Typical Command



After processing each command, lc_shell returns a status value that is the value of the command.

When you enter a long command with many options and arguments, you can split it across one or more lines by using the continuation character, backslash (\). For example,

```
lc_shell> read_lib "/paths/libraries \
/my_library.lib"
```

See the *Synopsys Logic Library Reference Manual* for the syntax and description of all the Library Compiler commands.

Redirecting Command Output

To send the output of an lc_shell command to a file instead of to standard output (usually the screen), use the > operator.

command > *file*

If the specified file does not exist, the tool creates it. If the file exists, it is overwritten.

For example, to redirect a report to a file,

```
lc_shell> report_lib my_lib1 > lib.report
```

To append command output to a file, use the >> operator.

command >> *file*

If the specified file does not exist, it is created. If the file does exist, the output is appended to the end of the file.

For example, to append a report to the end of a file, enter

```
lc_shell> report_lib my_lib2 >> lib.report
```

Using Commands

You can use lc_shell commands in the following two ways:

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

Getting Help Information

The Library Compiler tool has two levels of help information: command use and topic.

Command Use Help

The intended use (syntax) of an lc_shell command is displayed when you use the -help option with a command name. Command use displays the options and arguments for a command.

Topic Help

The `help` command displays information about an `lc_shell` command (including the `help` command itself).

Syntax

```
help [command_name]
```

command

Names the `lc_shell` command.

The `help` command lets you display the man pages interactively while you are running `lc_shell`. Man pages for all commands, variables, and predefined variable groups and user messages are included in the online help pages.

The following example returns the man page for the `help` command itself:

```
lc_shell> help
```

The following example returns the man page for the `report_lib` command:

```
lc_shell> help report_lib
```

If you request help for a command that cannot be found, the Library Compiler tool displays the following information message:

```
lc_shell> help xxxxx_topic
Information: No commands matched 'xxxxx_topic'. (CMD-040)
```

Reading In Libraries

The `read_lib` command loads a technology (.lib) source file into the Library Compiler tool and compiles it to a Synopsys database format (.db). When you load a .lib file, the Library Compiler tool checks out a Library Compiler license. If the license is missing, the library is still loaded, but all the functional information is removed; that is, all cells are black boxes and optimization is disabled. If the license is present, the Library Compiler tool releases the license when the `read_lib` command finishes loading the .lib file.

The `read_lib` command automatically performs basic syntax checks before it writes out the compiled library. If a required argument or attribute is missing, or if the syntax is incorrect, the Library Compiler tool issues a warning message or an error message, as applicable. In addition, the `read_lib` command automatically performs screener checks for the following data: timing, power, noise, scaling, multicorner-multimode, and IEEE 1801, also known as Unified Power Format (UPF).

To read and compile a library, such as a `cmos.lib` technology library, run the `read_lib` command at the Library Compiler prompt, as shown:

```
lc_shell> read_lib cmos.lib
```

For more information about the `read_lib` command and the `read_lib` library screener checks, see “Reading and Compiling Libraries” in the *Library Quality Assurance System User Guide*.

Listing Libraries

The `list_libs` command lists the names, file names, and paths of the libraries loaded in memory. To list the library information for one or more libraries, specify the library names with the `list_libs` command. If you do not specify the library names, all the libraries in memory are listed.

The library name is defined by the `library` group statement in the `.lib` file and can be different from the library file name.

To list the libraries loaded in memory, use the `list_libs` command, as shown:

```
lc_shell> list_libs my_lib
Logical Libraries:
Library      File          Path
-----      ----          -----
my_lib       my_lib.db   /synopsys/libraries
```

You can use the wildcard (*) and question mark (?) characters with the `list_libs` command.

Writing Library Files

The `write_lib` command saves a library memory file to a disk file in the Synopsys internal database (.db) format.

Syntax

```
write_lib library_name [-output file_name]
```

library_name

Specifies the name of the technology library to be written.

`-output file_name`

Specifies an output file name or path name for the library.

Examples

The following example writes the compiled CMOS technology library to a file called cmos.db. This file resides in the current working directory.

```
lc_shell> write_lib cmos
```

To write the same library to a file named cmos1.db in the current directory, enter

```
lc_shell> write_lib cmos -output cmos1.db
```

Generating Hierarchical .db Designs

The Library Compiler tool generates hierarchical .db designs with the `model` group, which can accept all the groups and attributes that a `cell` group accepts and the `cell_name` simple attribute.

A `model` group describes limited hierarchical interconnections. Currently, a `model` group is required only when shorted ports are present. When the `model` group is used, the Library Compiler tool writes out a separate model design .db file as an instantiation of the model cell plus the net connections for the shorted ports.

Syntax

```
model(model_name) {
    area : float;
    ...
    cell_name : cellname ; /*optional */
    ...
    pin(name1_string) {
        ...
    }
    pin(name2_string) {
        ...
    }
    short(name1, name2);
}
```

When the Library Compiler tool sees the `model` group, it generates

- A .db cell library called `cellname`, which is included in the library in memory
- A design wrapper called `model_name`, which instantiates the cell name

When you run the `write_lib` command, the Library Compiler tool saves the library .db files (lib.db) and all the generated model designs in one .db file (design.db).

Verifying Hierarchical .db Designs

The Library Compiler tool performs the following verification activities and issues a message if appropriate:

- Checks whether the `short` attribute is present. When it is missing, the Library Compiler tool issues an error message to the effect that this is a simple cell description and should not be processed as a model.
- Checks that the pin names listed in the `short` attribute statement are valid.
- Checks whether the `cell_name` attribute is present. When it is missing, the Library Compiler tool names the cell `model_name_core`.
- The Library Compiler tool issues an error message when it encounters both the `cell_name` attribute and the `short` attribute in a `cell` group.

The following example commands produce the .db files described:

Commands

```
lc_shell> read_lib mylib.lib
          (assume mylib.lib contains 5 models and 7 cells)
```

```
lc_shell> write_lib mylib
```

Files Produced

```
mylib.db           (contains a single library with 12 library cells)
```

```
mylib_model.db    (contains 5 model wrapper db designs)
```

Loading Compiled Libraries

You can load a previously saved .db library into the Library Compiler tool with the `read` command. The `read` command loads compiled libraries from a file into memory.

Syntax

```
read file_list
```

file_list

A list of one or more file names containing libraries.

For example, to load a compiled file named `lib.db`, enter

```
lc_shell> read lib.db
          Loading db file 'path/libraries/ex/lib.db'
```

To load multiple libraries, enter

```
lc_shell> read {my_lib1.db my_lib2.db}
Loading db file 'path/libraries/ex/my_lib1.db'
Loading db file 'path/libraries/ex/my_lib2.db'
```

Adding Library-Level Information to Existing Libraries

You can use the `set_attribute` command to add information at any level of an existing library.

Updating Attribute Values

This is the syntax for using the `set_attribute` command:

```
set_attribute {object_name, ...} attribute_name attribute_value
```

Note:

You can apply the `set_attribute` command only to libraries in which the library vendor has set the value of the `library_features` attribute to `allow_update_attribute`. For more information about using the `library_features` attribute, see the *Synopsys Logic Library Reference Manual*.

Example

The following example shows the values returned after executing the `set_attribute` command.

```
lc_shell> set_attribute [example/lib_udg1, example/lib/lib_udg2] \
           lib_udg_attr
new_value
Performing set_attribute on library object 'example/lib_udg1'
Performing set_attribute on library object 'example/lib/lib_udg2'
{"example/lib_udg1", "example/lib/lib_udg2"}
```

Accessing Library Information

The `get_attribute` command lets you access all the library attributes, including the user-defined attributes. However, you can access the user-defined attributes only when the library vendor sets the `library_features` value to `report_user_data`.

User-defined attributes are those attributes that library developers specify using the `define` and `define_group` attributes.

Accessing Simple Attribute Values

This is the syntax for using the `get_attribute` command to display values for simple attributes in an existing library:

```
get_attribute {object_name, ...} attribute_name
```

The following example shows the values returned after executing the `get_attribute` command.

Example

```
lc_shell> get_attribute {example/lib_udg1, example/lib/lib_udg2} \
           lib_udg_attr
Performing get_attribute on library object 'example/lib_udg1'
Performing get_attribute on library object 'example/lib/lib_udg2'
{"value1", "value_2"}
```

Reporting Library Information

The `report_lib` command displays information about technology libraries compiled in the Library Compiler database. The command displays timing, power, signal integrity, electromigration, and functionality information upon request. Cells are annotated according to the attributes specified for them. For more information about the `report_lib` command, see the “Generating Library Reports” chapter in the *Library Quality Assurance System User Guide*.

Some of the `report_lib` command options are as follows:

library

String specifying the name of the reported library. If the requested library is not in memory, an error message is printed.

`-timing`

Prints a report of timing information for verification. You can use the `-timing` option only when you read the .lib file into the Library Compiler tool during the same session.

`-timing_arcs`

Prints a list of the timing arcs defined for each cell.

`-timing_label`

Prints all the timing arc label information.

`-power`

Prints a report of power information for each cell that has power modeling.

-power_label

Prints all the power label information.

-em

Prints all information related to electromigration. This option applies only to technology libraries.

-table

Prints compact statetable information for technology library cells.

-full_table

Prints complete statetable and memory information for technology library cells.

-user_defined_data

Prints the values for user-defined attributes and groups if the library vendor has set the value of the `library_features` attribute to `report_user_data`. For more information about using the `library_features` attribute, see the *Synopsys Logic Library Reference Manual*.

-all

Prints all library information.

To redirect a library report to a disk file, use the > character, as shown in the following example.

```
lc_shell> report_lib -timing library >file_name
```

The following example displays information for the library, my_lib:

```
lc_shell> report_lib my_lib
```

When reporting on cells, the tool indicates the cell attributes:

`active_falling` and `active_rising`

This reports the active edge of the clock pin for flip-flops.

`active_high` and `active_low`

This reports the active level of the enable pin for latches.

`black box`

Because the Design Compiler tool cannot recognize what function the cell performs, it cannot insert the cell automatically into your design or replace it with gates in the netlist during synthesis.

clock_enable

This is the clock-enabling mechanism.

dont_touch

You set the `dont_touch` attribute to `true` in the `cell` group to indicate that all instances of the cell must remain in the network.

map_only

You set the `map_only` attribute to `true` in the `cell` group to exclude this cell from logic-level optimization during compilation.

preferred

You set the `preferred` attribute to `true` in the `cell` group to indicate that this cell is the preferred replacement during the gate-mapping phase of optimization.

removable

The Design Compiler tool knows what function the cell performs, so the cell can be replaced by another cell during synthesis and optimization. Even if some type of cell can be replaced, its function might be too complex for the Design Compiler tool to insert it into the design automatically.

sa0, sa1, sa01

You specified a stuck-at value in the `dont_fault` attribute for fault-modeling the cell or pin.

statetable

This cell is a register cell (flip-flop or latch).

dont_use

You set the `dont_use` attribute to `true` in the `cell` group to indicate that this cell should not be added to the design during optimization.

test_cell

This is a test cell.

Library Scaling

Library scaling is the interpolation of library data characterized at specific operating conditions to an intermediate operating condition. Library data interpolation eliminates the need for library characterization at the interpolated voltage and temperature. In voltage

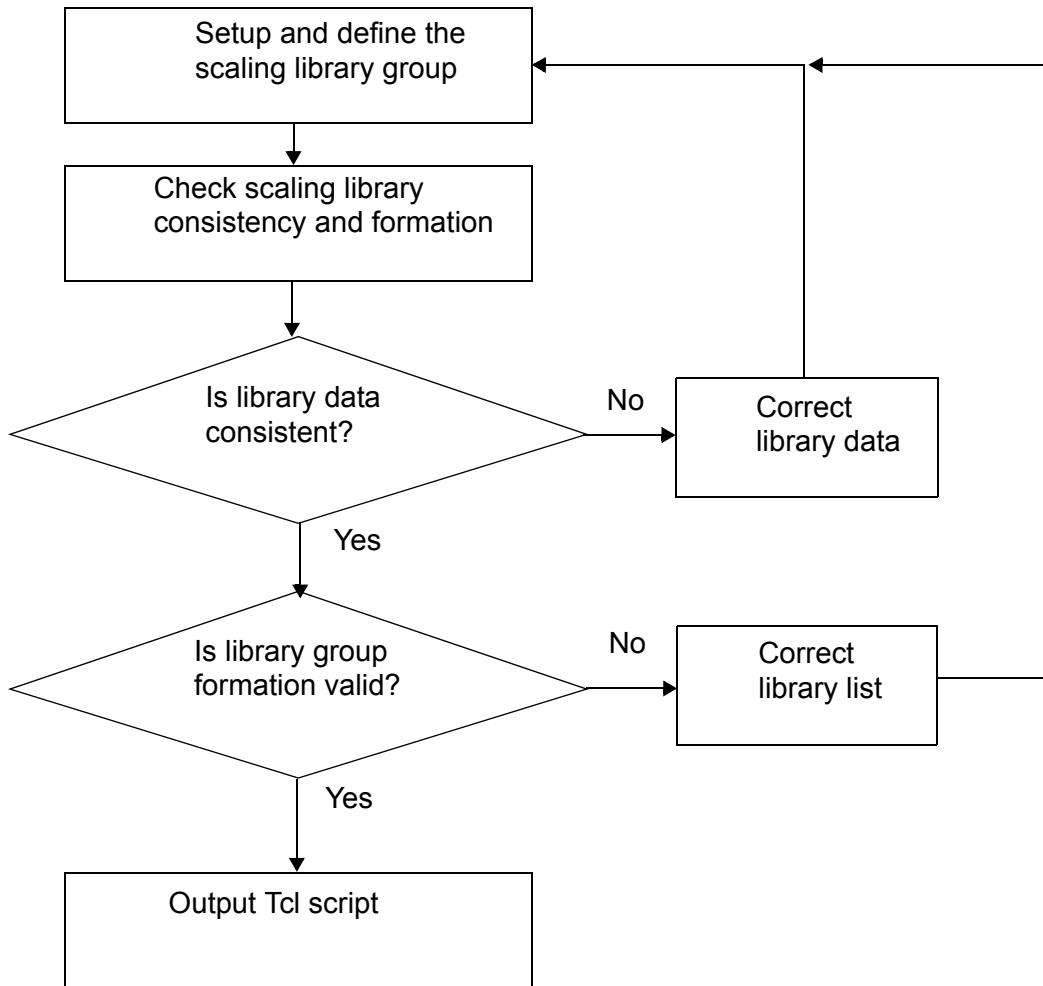
scaling, the libraries can be used at the interpolated voltage different from the library voltages. In temperature scaling, the libraries can be used at the interpolated temperature different from the library temperatures.

A scaling library group (or a scaling group) is a set of libraries grouped together for use in a scaling flow.

The following sections describe the library scaling flow and commands.

Library Scaling Flow

The library scaling flow enables you to find appropriate scaling groups for use in the scaling flows of other tools. The output is a Tool Command Language (Tcl) script containing valid scaling groups. After library characterization, use the library scaling flow to create scaling groups. [Figure 3-2](#) shows the library scaling flow diagram.

Figure 3-2 Library Scaling Flow

To define the scaling group, use the `create_scaling_lib_group` command.

To check the consistency of library data across all the libraries of the group and the scaling group formation (whether the group is fit for scaling), use the `check_library` command. The library consistency checks verify that the libraries are structurally consistent and have no missing models. The scaling group formation checks verify that the library voltage and temperature combinations meet the scaling requirements.

For more information about the library consistency checks and the scaling group formation checks, see the “Scaling Checks” section of the “Library Checking” chapter in the *Library Quality Assurance System User Guide*.

To output the scaling group in a Tcl file for use by other tools, specify the `write_scaling_lib_group` command.

Scaling Library Groups

You can generate two types of library groups with the scaling flow of [Figure 3-2](#):

- Valid scaling group

The valid scaling group meets the scaling group formation checks by having voltage and temperature combinations that form a grid. The libraries in this group also meet all the consistency check requirements for scaling.

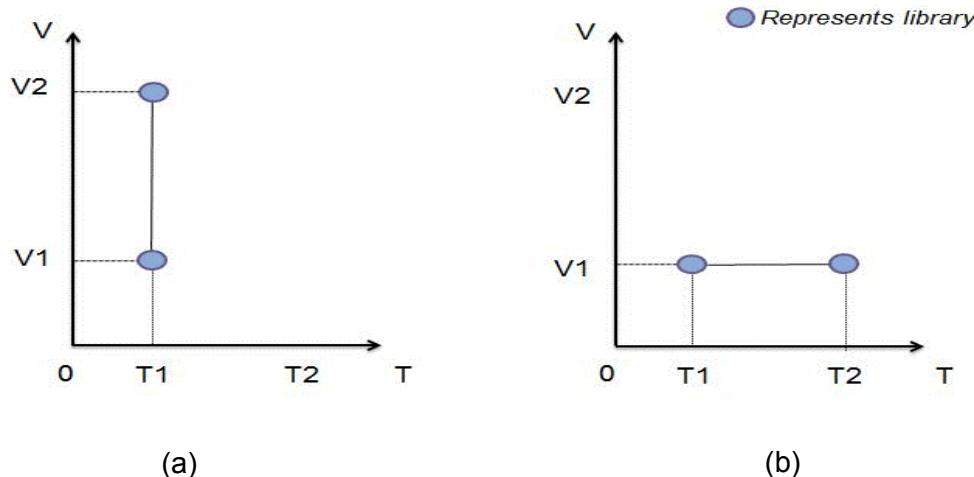
For a complete scaling group formation, a scaling group must have at least two libraries with the same temperature or voltage.

- Exact match group

The exact match group does not need to meet the group formation checks because library interpolation is not performed by other tools. The libraries are used at their characterized voltage and temperature. The libraries in this group only meet the consistency check requirements for scaling.

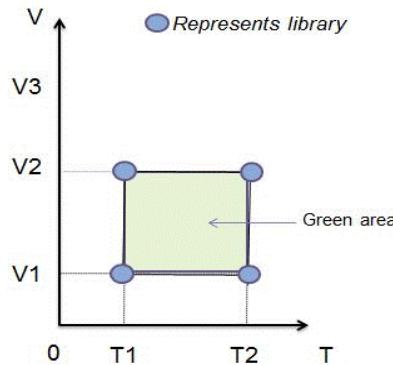
[Figure 3-3](#) shows two examples of scaling groups that meet the scaling group formation requirements. In part (a), the voltage can be interpolated between the voltages, V1 and V2, because the temperature, T1, is the same in both the libraries. In part (b), the temperature can be interpolated between the temperatures, T1 and T2, because the voltage, V1 is the same in both the libraries.

Figure 3-3 Minimum Requirement for Scaling Group Formation



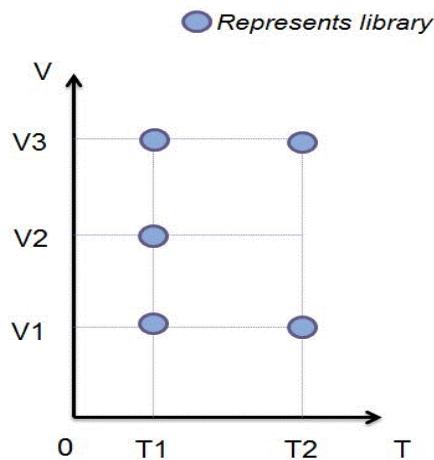
[Figure 3-4](#) shows a complete scaling group example with four libraries. The library voltages and temperatures form a complete grid and a valid scaling group formation. Library interpolation can occur anywhere in the highlighted area (in green) between the voltages, V1 and V2, and temperatures, T1 and T2.

Figure 3-4 A Complete Scaling Group Example



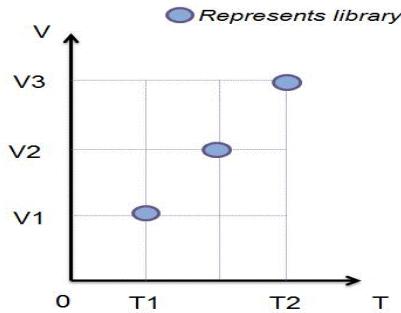
[Figure 3-5](#) shows a partial scaling group example with five libraries. The library voltages and temperatures form an incomplete grid. To complete and validate the scaling group formation, the missing library at the $V2T2$ corner needs to be added. You can use the partial scaling group for an exact match. For scaling, use a subset of the original group if it meets the formation requirements. To obtain the valid subsets or reduced groups, use the `write_scaling_lib_group -auto_adjust` command. In this example, the largest subset consists of the $V1T1$, $V2T1$ and $V3T1$ libraries.

Figure 3-5 A Partial Scaling Group Example



[Figure 3-6](#) shows an exact match group example. The group does not meet the requirement for a valid scaling group formation. There is no common voltage or temperature between the three libraries.

Figure 3-6 An Exact Match Group Example



Commands in the Library Scaling Flow

create_scaling_lib_group Command

The `create_scaling_lib_group` command defines the scaling group for voltage and temperature scaling.

Use the `-name` option to specify the scaling group name and then list the libraries that belong to the scaling group. Use the `-exact_match_only` option to create the exact match group where the scaling group formation check is not performed. Use the `-exclude_voltage_name` option to exclude a voltage name from scaling. By default, all the rail voltages are used in scaling.

check_library Command

The `check_library` command performs the library consistency and the scaling group formation checks on the libraries listed using the `create_scaling_lib_group` command.

For more information about the library consistency checks and the scaling group formation checks, see the Scaling Checks section of the “Library Checking” chapter in the *Library Quality Assurance System User Guide*.

write_scaling_lib_group Command

The `write_scaling_lib_group` command outputs a Tcl script for use by other tools and is specified in the scaling flow after the `check_library` command. The script includes the scaling group with valid library names.

Use the `-output` option with the `write_scaling_lib_group` command to specify the name of the output script file. Use the `-auto_adjust` option to obtain a library scaling subset or reduced group that meets the scaling group formation checks. This is useful when the

missing corner libraries cannot be found or a subset of the libraries meets the scaling requirements.

When you use the `-auto_adjust` option, the tool also revises the `link_library` variable value in the output script based on the subset scaling group. For example, if the original `link_library` variable is set to a library that does not belong to the reduced group, the tool replaces this library with a library from the reduced group. However, the original group name is retained even when the group is reduced.

Scaling Library Flow Examples

This section includes examples to generate scaling groups.

[Example 3-1](#) shows how to generate a complete scaling library group. The library temperature is the same in all the three libraries.

Example 3-1 Complete Scaling Library Group

```
lc_shell> set search_path "./path/to/db/libs"
lc_shell> set link_library "* 0p8v_25c.db"
lc_shell> create_scaling_lib_group -name grp1 \
    "0p8v_25c.db 1p0v_25c.db 1p2v_25c.db"
lc_shell> set_check_library_options -scaling {timing noise power}
lc_shell> check_library
lc_shell> write_scaling_lib_group -output slg.tcl
```

[Example 3-2](#) shows how to generate a partial scaling group for an exact match, meaning the libraries are used exactly as in their characterized voltage and temperature. The voltages and temperatures are different in all the three libraries.

Example 3-2 Partial Scaling Group for Exact Match

```
lc_shell> set search_path "./path/to/db/libs"
lc_shell> set link_library "* 0p8v_20c.db"
lc_shell> create_scaling_lib_group -name grp1 \
    "0p8v_20c.db 1p0v_50c.db 1p2v_80c.db" -exact_match_only
lc_shell> set_check_library_options -scaling {timing noise power}
lc_shell> check_library
lc_shell> write_scaling_lib_group -output slg.tcl
```

[Example 3-3](#) shows how to generate a complete scaling library group while checking only the library consistency on the specified cells.

Example 3-3 Specifying Cells for Library Consistency Checking

```
lc_shell> set search_path "./path/to/db/libs"
lc_shell> set link_library "* 0p8v_25c.db"
lc_shell> create_scaling_lib_group -name grp1 \
    "0p8v_25c.db 1p0v_25c.db 1p2v_25c.db"
lc_shell> set_check_library_options -scaling {timing noise power}
lc_shell> check_library -cells {INV DFF CGAT}
lc_shell> write_scaling_lib_group -output slg.tcl
```

[Example 3-6](#) shows the use of the `-exclude_voltage_name` option to exclude a voltage name from the scaling group formation check. The excluded voltage is a bias pin, VBP, that is specific to the library. This reduces the number of voltages to be scaled in the scaling dimensions.

Scaling dimensions are the total number of voltages and temperatures used for scaling. All the rail voltages that are defined using the `voltage_name` attribute are, by default, included in the scaling dimensions. All the nonzero voltages defined using the `voltage_map` attribute are, by default, used in the group formation checks.

Example 3-4 Excluding a Voltage From Scaling

```
lc_shell> set search_path    "./path/to/db/libs"
lc_shell> set link_library   "* 0p8v_25c.db"
lc_shell> create_scaling_lib_group -name grp1 "0p8v_25c.db \
1p0v_25c.db 1p2v_25c.db" -exclude_voltage_name VBP
lc_shell> set_check_library_options -scaling {timing noise power}
lc_shell> check_library
lc_shell> write_scaling_lib_group -output slg.tcl
```

[Example 3-5](#) shows the creation of an incomplete scaling group. The scaling group is incomplete because one library (required for the group to meet the scaling group formation checks) is missing from the defined scaling group. The defined scaling group does not pass the scaling group formation checks and the `check_library` command reports the valid subsets that meet the scaling group formation checks. The output script of the `write_scaling_lib_group` command includes the original scaling group excluding the valid subsets.

Example 3-5 Library Scaling Flow Without Using the -auto_adjust Option

```
lc_shell> set search_path    "./path/to/db/libs"
lc_shell> set link_library   "* 0p8v_25c.db"
lc_shell> create_scaling_lib_group -name grp1 "0p8v_25c.db \
1p0v_25c.db 1.2v_25c.db 0.8v_125c.db 1.2v_125c.db"
lc_shell> set_check_library_options -scaling {timing noise power}
lc_shell> check_library
lc_shell> write_scaling_lib_group -output slg.tcl
```

[Example 3-6](#) shows the creation of an incomplete scaling library group with a missing library. The `-auto_adjust` option is used with the `write_scaling_lib_group` to write out the best valid subset in the output script that meets the scaling group formation checks.

Example 3-6 Using the -auto_adjust Option

```
lc_shell> set search_path    "./path/to/db/libs"
lc_shell> set link_library   "* 0p8v_25c.db"
lc_shell> create_scaling_lib_group -name grp1 \
"0p8v_25c.db 1p0v_25c.db 1.2v_25c.db 0.8v_125c.db 1.2v_125c.db"
lc_shell> set_check_library_options -scaling {timing noise power}
lc_shell> check_library
lc_shell> write_scaling_lib_group -output slg.tcl -auto_adjust
```

[Example 3-7](#) shows the creation of a complete scaling library group. The `-no_link_library` option directs the `write_scaling_lib_group` command to write out the `link_library` variable value as a comment in the output Tcl script.

To use this script, you must define the `link_library` variable in other tools.

Example 3-7 Using the -no_link_library Option

```
lc_shell> set search_path "./path/to/db/libs"
lc_shell> set link_library "* 0p8v_25c.db"
lc_shell> create_scaling_lib_group -name grp1 \
    "0p8v_25c.db 1p0v_25c.db 1p2v_25c.db"
lc_shell> set_check_library_options -scaling {timing noise power}
lc_shell> check_library
lc_shell> write_scaling_lib_group -output slg.tcl -no_link_library
```

Understanding the `write_scaling_lib_group` Output

The output script of the `write_scaling_lib_group` command includes comments when the defined scaling group does not meet the scaling group checks. These messages are based on the corresponding LIBCHK error and warning tables reported by the `check_library` command. For use in the PrimeTime tool, this script must contain only the `link_library` variable and the `create_scaling_lib_group` command specification. You must correct the reported issues.

LIBCHK 303, 304, and 306 Messages

The following example shows an input script that generates the LIBCHK-303, 304, and 306 messages in the output script. The input script is:

```
lc_shell> create_scaling_lib_group -name GRP1 "0p63vn40c.db \
    0p81v125c.db 0p81vn0c.db 0p9v25c.db \
    0p9v85c.db 0p99v125c.db 0p99vn40c.db"
lc_shell> set_check_library_options -scaling {timing power noise}
lc_shell> check_library
lc_shell> write_scaling_lib_group -output slg.out.tcl
```

The output script is:

```
#link library has extra library (LIBCHK-306).
set link_library "0p63vn40c.db 0p81v125c.db 0p81vn0c.db 0p9v25c.db
0p9v85c.db 0p99v125c.db 0p99vn40c.db"
#Add missing corner libraries to group GRP1 (LIBCHK-303) or use reduced
#groups (LIBCHK-304).
create_scaling_lib_group -name GRP1 "0p63vn40c.db 0p81v125c.db
0p81vn0c.db 0p9v25c.db 0p9v85c.db 0p99v125c.db 0p99vn40c.db"
```

The input script has an incomplete scaling group (GRP1) with missing corner libraries. Add the missing libraries as indicated by the LIBCHK-303 message, or use a valid reduced group

as indicated by the LIBCHK-304 message. The valid reduced groups are listed in the LIBCHK-304 section of the log file that begins with the following information:

```
Information: Valid scaling library groups (LIBCHK-304)
```

The LIBCHK-306 message indicates that the `link_library` variable has multiple libraries specified. You must specify only one library from the scaling group with the `link_library` variable.

LIBCHK-304 Message

The following example shows an input script that generates the LIBCHK-304 message in the output script. The input script is:

```
lc_shell> set link_library "lib1_0p63vn40c.db"
lc_shell> create_scaling_lib_group -name SVT \
    "lib1_0p63vn40c.db lib2_0p99v0c.db lib3_0p99v85c.db"
lc_shell> set_check_library_options -scaling {timing power noise}
lc_shell> check_library
lc_shell> write_scaling_lib_group -output slg.out.tcl -auto_adjust
```

The output script is:

```
set link_library "lib1_0p63vn40c.db"
#No valid library group found from SVT meeting formation (LIBCHK-304)
```

In the input script, an incomplete scaling group is defined. Because the scaling group does not meet the scaling group formation requirements, the `-auto_adjust` option is used with the `write_scaling_lib_group` command to write out the valid subset or reduced scaling groups. However, the `write_scaling_lib_group -auto_adjust` command does not find a valid subset scaling group. The LIBCHK-304 message indicates that valid reduced scaling groups do not exist. The missing libraries are listed in the LIBCHK-303 section of the log file that begins with the following information:

```
Warning: Missing libraries in scaling groups (LIBCHK-303)
```

LIBCHK-306 Message

The following example shows an input script that generates the LIBCHK-306 message in the output script. The input script is:

```
lc_shell> set link_library "lib4_0p99v0c.db"
lc_shell> create_scaling_lib_group -exact_match_only -name SVT \
    "lib1_0p99v0c.db lib2_0p99v125c.db lib3_0p99vn40c.db"
lc_shell> set_check_library_options -scaling {timing power noise}
lc_shell> check_library
lc_shell> write_scaling_lib_group -output slg.out.tcl
```

The output script is:

```
#link_library is missing a library from group 'SVT' (LIBCHK-306).
set link_library " lib4_0p99v0c.db "
```

```
create_scaling_lib_group -name SVT " lib1_0p99v0c.db lib2_0p99v125c.db
lib_3_0p99vn40c.db
```

In the input script, the `link_library` variable does not specify a library from the group, SVT. The scaling group meets the exact match requirements. The LIBCHK-306 message indicates that the `link_library` value is not valid. In the input script, you must set the `link_library` variable value to one of the three libraries of the SVT group.

LIBCHK-362 Message

The following example shows an input script that generates the LIBCHK-362 messages in the output script. The input script is:

```
lc_shell> set link_library "lib1_0p99v0c.db"
lc_shell> create_scaling_lib_group -exact_match_only -name SVT \
"lib1_0p99v0c.db lib2_0p99v125c.db lib3_0p99vn40c.db"
lc_shell> set_check_library_options -scaling {timing power noise}
lc_shell> check_library
lc_shell> write_scaling_lib_group -output slg.out.tcl
```

The output script is:

```
set link_library "lib1_0p99v0c.db"
#library group SVT consistency check failed (LIBCHK-362)
create_scaling_lib_group -name SVT "lib1_0p99v0c.db lib2_0p99v125c.db
lib3_0p99vn40c.db" -exact_match_only
```

The libraries do not meet the data consistency checks. The LIBCHK-362 message indicates that the libraries have inconsistencies for scaling. So, the scaling group does not meet the exact match requirement. The log file shows the following warning and information messages. You must make the library data consistent.

```
Warning: Library 'lib1_0p99v0c.db (lib#1)' is missing CCS driver models. (LIBCHK-332)
Warning: Library 'lib1_0p99v0c.db (lib#1)' is missing CCS receiver models. (LIBCHK-332)
Warning: Library 'lib1_0p99v0c.db (lib#1)' is missing CCS noise models. (LIBCHK-332)
Warning: Library 'lib2_0p99v125c.db (lib#2)' has no power data for power scaling.
Information: Logic library inconsistencies found for timing scaling. (LIBCHK-362)
Information: Logic library inconsistencies found for noise scaling. (LIBCHK-362)
Information: Logic library inconsistencies found for power scaling. (LIBCHK-362)
```

Removing Libraries From Memory

Use the `remove_lib` command to remove libraries from lc_shell memory. The reclaimed memory is then available for other libraries.

Note:

To save a changed library to a disk file, use the `write_lib` command to save that library to a UNIX file before using the `remove_lib` command.

Syntax

```
remove_lib [lib_list] [-all]
```

If you do not define any arguments, no library is removed.

lib_list

List of libraries to be removed. Separate each library name with a space.

-all

Removes all libraries in memory.

Note:

The `remove_lib` command does not free swap space (memory) already claimed by the Library Compiler tool. It does, however, reuse reclaimed memory for future `lc_shell` commands.

Using Shell Commands

The following sections explain different shell commands to use with `lc_shell`.

Command Aliases

Some `lc_shell` commands with options and arguments can be quite long. The `alias` command creates a shortcut (or alias) for commands that you use frequently.

For the Library Compiler tool to recognize an alias, use the `alias` command at the beginning of a line.

An alias definition takes effect immediately but lasts only until you exit the Library Compiler work session. To save commonly used alias definitions, include them in the `.synopsys_lc.setup` (or other setup) file. See “[Using Setup Files](#)” on page 3-5 for more information about using `.synopsys_lc.setup` and other setup files.

The `unalias` command removes alias definitions.

alias Command

Instead of typing in long command strings, you can use the `alias` command to create shortcuts for commands you use frequently.

Syntax

```
alias [identifier [definition]]
```

identifier

The name of the alias you are creating (if *definition* is supplied) or listing (if *definition* isn't supplied). The name can contain letters, digits, and underscores (_). If no *identifier* is given, all aliases are listed.

definition

The commands for which you are creating an alias. If the *identifier* is already defined, the definition overwrites the existing definition. If you do not define *definition*, lc_shell displays the definition of *identifier*.

Example

In the example, the `alias` command creates an alias called `wlo`.

```
lc_shell> alias wlo "write_lib -output"
```

After creating the `wlo` alias, you can write a library by only entering the alias, as shown in the next example. (When you have a library called `lib1` already loaded into lc_shell.)

Example

```
lc_shell> wlo lib1.db
```

The lc_shell program does not echo the entered alias definition.

unalias Command

The `unalias` command removes alias definitions.

Syntax

```
unalias [patterns...]
```

patterns ...

A list of one or more aliases whose definitions you want removed.

Examples

This example shows how to remove the `wlo` alias.

```
lc_shell> unalias wlo
```

This example shows how to remove the aliases beginning with `w`.

```
lc_shell> unalias w*
```

This example shows how to remove all aliases created by the `alias` command.

```
lc_shell> unalias *
```

Listing Previously Entered Commands

To list the commands you have used in the `lc_shell` session, use the `history` command. By redirecting the output of the `history` command, you can create a file to use as the basis for a command script, as shown in the examples in this section.

[“Reexecuting Commands” on page 3-32](#) explains how to reexecute a previous command.

Syntax

```
history [n] [-r] [-h]
```

n

Lists up to *n* previously entered commands.

-r

Lists the commands in reverse order. The most recent command is listed first.

-h

Displays commands without index numbers (commands are numbered in each session). This option is useful in creating a command script file from previously entered commands.

For example, to display a list of all commands entered in the session, enter

```
lc_shell> history
```

To list only the previous ten commands, enter

```
lc_shell> history 10
```

To create a command script file from a history of commands, use the `history -h` command and redirect the output to the desired script file.

```
lc_shell> history -h > history.script
```

Note:

You must edit the created file (`history.script`), because the last line contains the call to `history` itself and overwrites the file.

To execute the new script file later, use the `source` command or start `lc_shell` with the `-f` option.

```
lc_shell> source history.script
```

Reexecuting Commands

To reexecute previously entered commands (command substitutions), use the exclamation point (!) operator:

!!

Reexecutes the last command.

!-n

Reexecutes the *n*th command from the last.

!n

Reexecutes the command numbered *n* (from a history list).

!text

Reexecutes the most recent command that started with *text*. The text must begin with a letter or an underscore (_) and can contain numbers.

!?text

Reexecutes the most recent command that contains *text* anywhere in it. The text must begin with a letter or an underscore (_) and can contain numbers.

When an lc_shell command begins with an exclamation point, the indicated command is echoed in its entirety.

```
lc_shell> !!
last_command
```

If you enter the following commands,

```
lc_shell> read_lib cmos.lib
lc_shell> write_lib cmos
```

you can reexecute the last command (`write_lib`) and add an option; for example, the `-output` option.

```
lc_shell> !! -output cmos1.db
write_lib cmos -output cmos1.db
```

To list the command history, enter

```
lc_shell> history
1 read_lib cmos.lib
2 write_lib cmos
3 write_lib cmos -output cmos1.db
4 history
```

To reexecute the previous `read_lib` command (command number 1), enter

```
lc_shell> !1  
read_lib cmos.lib
```

To reexecute the last command starting with “re”, enter

```
lc_shell> !re  
read_lib cmos.lib
```

To reexecute the last command containing “output”, enter

```
lc_shell> !?output  
write_lib cmos -output cmos1.db
```

Shell Commands

The command interface includes these four commands that communicate with the system environment:

`pwd`

Lists the Library Compiler working directory.

`cd directory`

Changes the Library Compiler working directory to the specified directory or to your home directory if no directory is specified.

`ls directory`

Lists the files in the specified directory or in the working directory if no directory is specified.

`sh "command"`

Calls the UNIX operating system to execute the specified command. The command must be enclosed in quotation marks.

[Example 3-8](#) shows how to use the four shell commands, starting by invoking the `lc_shell` program from a system prompt.

Example 3-8 Using Shell Commands

```
prompt> cd  
prompt> pwd  
/path  
prompt> lc_shell  
Library Compiler (TM)  
...  
Initializing...  
lc_shell> pwd
```

```
/path
lc_shell> cd my_designs
lc_shell> pwd
/path/my_designs
lc_shell> ls
my_lib1.db
my_lib2.db
my_libs.script

lc_shell> sh "more my_designs.script"
read /path/my_designs/
my_design1.db
...
...
```

Command Scripts

A script file, also called a command script, is a sequence of lc_shell commands in a text file. Use the `source` command to execute the commands in a script file.

By default, the Library Compiler tool displays commands in the script file as they execute. The `echo_include_commands` variable in the system variable group, determines if included commands are displayed as they are processed. The default for the `echo_include_commands` variable is `true`.

You can write comments in your script file by using the /* and */ delimiters.

Using the source Command

You use the `source` command to execute scripts in lc_shell.

Syntax

`source file`

`file`

Name of the script file. If `file` is a relative path name (it does not start with `/`, `./`, `../`, or `~`), the Library Compiler tool searches for the file in the directories listed in the `search_path` variable (in the system variable group). The tool reads the file from the first directory in which it exists.

For example, to execute the commands contained in the `my_script` file in your home directory, enter

```
lc_shell> source home-directory/my_script
command
command
/* comment */
...
...
```

To execute the commands in the `example.script` file, where the path to this file is defined by the `search_path` variable, enter

```
lc_shell> set search_path {., my_dir, /path/libraries}
{., my_dir, /path/libraries}
lc_shell> source my_script.tcl
command
command
/* comment */
. . .
```

In the previous example, the Library Compiler tool searches for the `my_script` file first in the current directory (.), then in the subdirectory `my_dir`, then in the directory `/path/libraries`.

Using Control Flow Commands

Control flow commands determine the execution order of other commands. Three control flow commands are `if`, `while`, and `foreach`.

An `if` statement contains several sets of commands. One set of these commands is executed one time, as determined by the evaluation of a given condition expression.

The `while` command repeatedly executes a single set of commands, so long as a given condition evaluates true.

The `foreach` command executes a set of commands one time for each value of a defined variable.

You can use any of the `lc_shell` commands in a control flow command, including another `if`, `while`, or `foreach` command.

Primarily, you use the control flow commands in command scripts. A common use is to check if a previous command executed successfully.

The condition expression is evaluated as a Boolean variable. [Table 3-2](#) shows how each non-Boolean variable type is evaluated. For example, the integer 0 is evaluated as a Boolean false. A nonzero integer is evaluated as a Boolean true. Use condition expressions

to compare two variables of the same type or a single variable of any type. All variable types have Boolean evaluations.

Table 3-2 Boolean Evaluations of Non-Boolean Types

Boolean evaluation	Integer/ floating-point number	String	List
False	0, 0.0	" "	{ }
True	Others	Nonempty string	Nonempty list

if Command

The `if` command executes when the specified expression is true.

Syntax

```
if (if_expression) {
    if_command
    if_command
    ...
} else if (else_if_expression) {
    else_if_command
    else_if_command
    ...
} else {
    else_command
    else_command
    ...
}
```

This is how the `if` command is executed:

1. When `if_expression` is true, execute all `if_commands`.
2. When `if_expression` is not true and `else_if_expression` is true, execute all `else_if_commands`.
3. When neither `if_expression` nor `else_if_expression` is true, execute all `else_commands`.

Each expression is evaluated as a Boolean variable (see [Table 3-2](#)).

The `else_if` and `else` branches are optional. If you use an `else_if` or `else` branch, the word `else` must be on the same line as the preceding right brace (`}`), as shown in the syntax. You can have many `else_if` branches but only one `else` branch.

The following script example shows how to use a variable (`vendor_library`) to control the link and font libraries.

```
vendor_library = my_lib
if ((vendor_library != Xlib) && (vendor_library
!= Ylib)) {
    vendor_library = Xlib
}
if (vendor_library == Xlib) {
    link_library = Xlib.db
    font_library = l_25.font
} else {
    link_library = Ylib.db
    font_library = l_30.font
}
```

while Command

The `while` command repeats a command between the `while` and the matching `end` statement when the expression is true.

Syntax

```
while (expression) {
    while_command
    while_command
    ...
}
```

If `expression` is true, the Library Compiler tool executes all `while` commands repeatedly. The Library Compiler tool evaluates the expression as a Boolean variable (see [Table 3-2](#)).

If the Library Compiler tool encounters the `continue` command in a `while` loop, command execution immediately starts over at the top of the `while` loop.

If the Library Compiler tool encounters the `break` command, command execution jumps to the next command after the end of the `while` loop. Continued command execution depends on successful completion of the previous command.

foreach Command

The `foreach` command executes a set of commands one time for each value assigned.

Syntax

```
foreach (variable_name, list ) {
    foreach_command
    foreach_command
    ...
}
```

variable_name

The name of the variable successively set to each value in *list*.

list

Any valid expression containing variables or constants.

foreach_command

An lc_shell statement. All statements must be terminated by either a semicolon or a carriage return.

The `foreach` command sets *variable_name* to each value represented by *list* and executes the identified set of commands for each value. The *variable_name* retains its value upon termination of the `foreach` loop.

The following example shows a simple case of traversing all the elements of *list* variable *x* and displaying them:

```
x = {a, b, c}
foreach ( member, x ) {
    list member ;
}
```

The result of this command is

```
member = "a"
member = "b"
member = "c"
```

continue Command

The `continue` command is used only in a `while` or `foreach` command. The `continue` command causes the programs to skip the remainder of the loop's commands, begin again, and reevaluate *expression*. If *expression* is true, all commands are executed again.

Syntax

```
continue
```

break Command

The `break` command is used only in a `while` or `foreach` command. The `break` command causes the program to skip the remainder of the loop's commands and jump to the first command outside the loop.

Syntax

```
break
```

Note:

The difference between the `continue` and `break` commands is that the `continue` command causes command execution to start over, whereas the `break` command causes command execution to break out of the `while` or `foreach` loop.

Using the Wildcard Character

You can use the wildcard character, an asterisk (*), to match a number of characters in a name. For example,

`u*`

Matches all object names that begin with the letter *u*.

`u*z`

Matches all object names that begin with the letter *u* and end in the letter *z*.

`*`

Double backslashes (\ \) that precede a wildcard character remove the special meaning of the wildcard character.

Using Variables

Variables store values for use by `lc_shell` commands. You can set variable values in the `lc_shell` interface. Variables can be a character, an integer, a floating-point number, or list data.

Each `lc_shell` variable has a *name* and a *value*.

name

A sequence of characters. Some variables, such as `command_log_file` (the name of the `lc_shell` log file) and `text_print_command` are predefined.

value

A file name or list of file names, a number, or a set of command options and arguments.

A variable can be part of a variable group. The following sections explain how to set, create, list, and remove variables.

Setting Variable Values

To set the value of a variable, enter the `set` command, the variable name, and the appropriate value. The Library Compiler tool echoes the new value.

```
lc_shell> set search_path { . /path/libraries}  
{ ., /path/libraries}
```

You can also set a variable to be equal to a list of library objects.

Creating Variables

To create a new variable, use the `variable` command. A value can be a number, a string, or a path as shown in the examples.

```
lc_shell> variable numeric_var 107.3  
lc_shell> variable my_var my_value  
lc_shell> variable list_var{/home/frank, /usr/libraries, /tmp}
```

Listing Variable Values

To display a variable value, use the `printvar` command.

```
lc_shell> printvar numeric_var  
numeric_var = "107.3"  
  
lc_shell> printvar my_var  
my_var = "my_value"
```

Removing Variables

To remove a variable from `lc_shell`, use the `unset` command.

```
lc_shell> unset my_var
```

Variables required by `lc_shell` cannot be removed. For example, you cannot remove system variables, such as `command_log_file` and `search_path`.

Productions

Syntax is described in terms of terminals

Terminals are written in Courier plain font and represent text that must be entered as shown.

nonterminals

Nonterminals represent text that you can enter in several different ways, as defined by the productions).

Each production begins with the name of a nonterminal followed by one or more lines that start with ::= (colon, colon, and equal sign). The text that follows the ::= represents one way to enter the nonterminal. This representation is based on the Backus-Naur Form (BNF).

Brackets ([]) around a collection of terminals or nonterminals indicate that the enclosed items are optional; that is, the brackets indicate zero or one instance.

Braces ({ }) around a collection of terminals or nonterminals indicate that the enclosed items are optional and can be repeated; that is, the braces indicate zero or more instances.

Note:

The nonterminal digit refers to the characters 0 through 9. The nonterminal alphabetic refers to the characters a through z and A through Z.

Syntax

```
statement      ::=      simple_statement { ; simple_statement } [;]

simple_statement ::=      command_statement [ output_redirect ]
                     ::=      assignment_statement [ output_redirect ]
                     ::=      control_statement [ output_redirect ]

command_statement ::=      command_name options arguments

output_redirect  ::=      > file_name
                     ::=      >> file_name

assignment_statement ::=      variable_name = expression

control_statement  ::=      if control_expression {
                                statement ;
                            }
                           ::=      if control_expression {
                                statement ;
                            }
                           ::=      else {
                                statement ;
                            }
                           ::=      if control_expression {
                                statement ;
                            }
                           ::=      else if control_expression {
                                statement ;
                            }
                           ::=      else {
                                statement ;
                            }

                           ::=      while control_expression {
                                statement ;
                            }

                           ::=      foreach (variable_name, list) {
                                statement ;
                            }

                           ::=      continue
```

```

        ::=      break

options      ::=      -identifier
                ::=      -identifier argument

argument     ::=      [ () expression_list ()]

expression_list ::=      expression { [,] expression }

expression    ::=      string_expression
                ::=      numeric_expression
                ::=      variable_expression
                ::=      list_expression
                ::=      command_expression
                ::=      control_expression
                ::=      logic_expression
                ::=      operator_expression
                ::=      ( expression )

string_expression ::=      " {character} "
                    ::=      string_expression & string_expression

numeric_expression ::=      [digit] {digit}
                ::=      {digit} . {digit} [ e [unary_operator] digit
                                {digit} ]

variable_expression ::=      identifier

list_expression ::=      { [ expression { [,] expression } ] }

command_expression ::=      command_name ( list_expression )

control_expression ::=      ( logic_expression relational_operator
                                logic_expression )

logic_expression ::=      expression
                ::=      ! ( control_expression )
                ::=      ( control_expression ) && ( control_expression )
                ::=      ( control_expression ) || ( control_expression )

operator_expression ::=      expression binary_operator expression
                            unary_operator expression

```

```
relational_operator ::= ==  
                      ::= !=  
                      ::= >  
                      ::= >=  
                      ::= <  
                      ::= <=
```

Note:

The relational operators `>`, `>=`, `<`, and `<=` bind the most tightly (all at equal precedence), then `==`, and `!=`.

```
binary_operator ::= +  
                  ::= -  
                  ::= *  
                  ::= /  
  
unary_operator ::= +  
                  ::= -  
  
file_name      ::= expression  
  
variable_name   ::= identifier  
  
command_name    ::= identifier  
  
identifier      ::= first_id_character { rest_id_character }
```

```
first_id_character   ::=      alphabetic
                      ::=      '
                      ::=      #
                      ::=      ~
                      ::=      \
                      ::=      °
                      ::=      §
                      ::=      &
                      ::=      ^
                      ::=      @
                      ::=      !
                      ::=      -
                      ::=      [
                      ::=      ]
                      ::=      |
                      ::=      ?
                      ::=      *
                      ::=      -
                      ::=      +
                      ::=      /
                      ::=      .
                      ::=      :
                      ::=      rest_id_character
rest_id_character    ::=      digit
                      ::=      first_id_character
```


4

Using Library Compiler Syntax

The Library Compiler tool compiles a technology library. The technology library stores library characterization values using the syntax consisting of groups, attributes, definitions, and comments. To build a library, you must understand the following concepts explained in this chapter:

- [Library File Names](#)
- [Library, Cell, and Pin Names](#)
- [Statements](#)
- [Comments](#)
- [Units of Measure](#)
- [Library Example](#)

Library File Names

File naming conventions are not predefined in the Library Compiler tool. Synopsys uses the following suffixes for library file names; use these suffixes to simplify file maintenance:

.lib

Technology library source files

.db

Compiled technology libraries in Synopsys database format

Library, Cell, and Pin Names

The following rules apply to naming conventions for libraries:

- Use the same name for libraries, cells, and pins as the vendor uses in its libraries.
- Names are case-sensitive. For example, the name AND2 is not the same as And2.
- The Library Compiler tool accepts names of any length.
- Enclose names that do not begin with an alphabetic character in quotation marks (" "). For example, express a cell named 2NAND as "2NAND". This rule has exceptions; see the sections that describe the `function` attribute and the `related_pin` attribute in the *Synopsys Logic Library Reference Manual*.
- Each identifier must be unique within its own context. The Library Compiler tool flags each duplicate cell name in a library as a warning and flags each duplicate pin name in a cell as an error.

Statements

Statements are the building blocks of a library. Enter all library information with one of the following types of statements:

- Group statements
- Simple attribute statements
- Complex attribute statements
- Define statements

You can enter a statement on multiple lines. A continued line must end with a backslash (\).

Group Statements

A group is a collection of statements that defines a library, a cell, a pin, a timing arc, and so forth. A pair of braces ({}) encloses the contents of the group.

This is the general syntax of a group statement:

Syntax

```
group_name (name) {  
    ... statements ...  
}
```

name

A string that identifies the group. Check the individual group statement syntax definition to verify if *name* is required, optional, or null. You must type the group name and the right brace symbol ({) on the same line.

[Example 4-1](#) defines pin group A.

Example 4-1 Group Statement Specification

```
pin(A) {  
    ... pin group statements ...  
}
```

Attribute Statements

An attribute statement defines characteristics of specific objects in the library. Attributes applying to a specific object are assigned within a group statement defining the object.

In this user guide, the word *attribute* refers to all attributes unless specifically stated otherwise. Attributes can be of two types: simple and complex.

Unless mentioned otherwise, use an attribute only one time within a group statement. If an attribute is used multiple times, the Library Compiler tool recognizes only the last attribute statement.

Simple Attributes

Simple attributes have the following syntax:

Syntax

```
attribute_name : attribute_value ;
```

You must separate the attribute name from the attribute value with a space, colon, and another space. Place the attribute statement in a single line. The semicolon at the end of the line is optional. Use the Return key to start a new line.

[Example 4-2](#) adds a `direction` attribute to the `pin` group in [Example 4-1](#).

Example 4-2 Simple Attribute Specification

```
pin(A) {  
    direction : output ;  
}
```

For certain simple attributes, you must enclose the attribute value in quotation marks, as shown here:

```
attribute_name : "attribute_value" ;
```

For certain simple attributes, you must enclose the attribute value in quotation marks.

[Example 4-3](#) adds the function `X + Y` to the pin example.

Example 4-3 Defining the Function of a Pin

```
pin (A) {  
    direction : output ;  
    function : "X + Y" ;  
}
```

Complex Attributes

A complex attribute statement includes one or more parameters enclosed in parentheses. (The brackets denote optional parameters.)

Syntax

```
attribute (parameter1 [, parameter2, parameter3 ...] );
```

The following example uses the `mode` complex attribute to define the mode name and the current mode value of a library cell.

```
mode (rw, read);
```

Define Statements

The `define_group` and `define` statements let you create new groups and attributes.

Defining New Groups

The `define_group` statement lets you create new (user-defined) groups.

Syntax

```
define_group (group_name_id, parent_name_id, );
```

group_name

The name of the new group you are creating.

parent_name

The name of the group in which this attribute is specified.

For example, to define a new group called `myGroup`, which is valid in a `pin` group, use

```
define_group (myGroup, pin) ;
```

Defining New Attributes

The `define` statement lets you create new simple attributes (user-defined). The syntax for `define` statements is similar to the syntax for complex attributes.

Syntax

```
define (attribute_name, group_name, attribute_type);
```

attribute_name

The name of the new attribute you are creating.

group_name

The name of the group in which this attribute is specified.

attribute_type

The type of values you want to allow for the new attribute, such as `string`, `integer`, or `float` (for floating-point number).

You can use either a space or a comma to separate the arguments. For example, to define a new string attribute called `mystring`, which is valid in a `pin` group, you can use

```
define (mystring, pin, string) ;
```

You give the new attribute a value by using the simple attribute syntax:

```
mystring : "nimo" ;
```

Comments

Comments explain library entries. Although never required, liberal use of comments gives other users the information they need to understand your entries. Enclose comments between the /* and */ delimiters.

```
/* This is a one-line comment. */

/* This is a
   multiline comment. */
```

The Library Compiler tool ignores all characters between the delimiters.

Units of Measure

Downstream tools require that a design use the same units of measure as those used in the library. For example, if the library unit of time is in nanoseconds, you must enter all timing values in nanoseconds. If picofarads are the units of capacitance in the library, you must enter all capacitance values in picofarads. All units and entries in a library must be consistent to ensure accurate timing calculation.

The Library Compiler tool supports four discrete time units: 1 ns, 100 ps, 10 ps, and 1 ps. For more information about time units, see “[library Group](#)” on page 5-3.

You can optionally supply a voltage unit, current unit, resistance unit for pull ups and pull downs, capacitive load unit, and power unit. For more information, see “[Input-Capacitance Characterization Attributes](#)” on page B-14.

For area, use equivalent gates or some other internal metric (The ASIC vendor determines the measurement units of area). Ensure that you use the same unit of measure throughout the library. This area consistency requirement also applies to the estimation of wire area. [Table 4-1](#) shows commonly used measurement units.

Table 4-1 Commonly Used Units of Measure

Component	Unit of measure
Area	Equivalent gates
	Square microns for standard cells
	Cell-based custom designs
Capacitance	Picofarads
	Standardized loads
Delay	Nanoseconds

Table 4-1 Commonly Used Units of Measure (Continued)

Component	Unit of measure
Resistance	Kilohms
Temperature	Degrees centigrade
Voltage	Volts

Library Example

The following example shows a CMOS technology library describing two cells: AN2 and OR2.

```
library (example) { /* library group till end*/
  technology (cmos) ; /* complex attribute */
  delay_model : table_lookup ;
  /*2-input AND gate*/
  cell (AND2) { /* cell group */
    pin (A, B) {
      direction : input ;
      capacitance : 1 ;
    }
    pin (Z) { /* pin group */
      direction : output ;
      function : "A * B" ;
      timing () { /* timing group */
        cell_rise(scalar) {
          values ( "0.1");
        }
        rise_transition(scalar) {
          values ( "0.1");
        }
        cell_fall(scalar) {
          values ( "0.1");
        }
        fall_transition(scalar) {
          values ( "0.1");
        }
        related_pin : "A B" ;
      } /* End timing group */
    } /* End pin group */
  } /* End cell group */

  /*2-input OR gate*/
  cell (OR2) {
    pin (A, B) {
      direction : input ;
    }
  }
}
```

```
    capacitance : 1 ;
}
pin (Z) {
    direction : output ;
    function : "A + B" ;
    timing () {
        cell_rise(scalar) {
            values ( "0.1");
        }
        rise_transition(scalar) {
            values ( "0.1");
        }
        cell_fall(scalar) {
            values ( "0.1");
        }
        fall_transition(scalar) {
            values ( "0.1");
        }
        related_pin : "A B" ;
    }
}
} /* End library group */
```

5

Building a Technology Library

A library description identifies the characteristics of a technology library and the cells it contains. The `library` group contains the entire library description. This chapter describes the `library` group-level attributes of a CMOS technology library. The following concepts and tasks are explained in this chapter:

- [Creating Library Groups](#)
- [Using General Library Attributes](#)
- [Documenting Libraries](#)
- [Delay and Slew Attributes](#)
- [Defining Units](#)
- [Describing Pads](#)
- [Describing Power](#)
- [Using Processing Attributes](#)
- [Setting Minimum Cell Requirements](#)
- [Setting CMOS Default Attributes](#)

Creating Library Groups

The `library` group contains the entire library description. Each library source file must have only one `library` group. Attributes that apply to the entire library are defined at the `library` group level, at the beginning of the library description.

For information about the default attributes and scaling factors, see [Chapter 6, “Building Environments](#). For information about library-level group statements, see [Chapter 7, “Defining Core Cells](#).

Syntax

[Example 5-1](#) shows the general syntax of the library. The first statement names the library. The following statements are library-level attributes that apply to the library as a whole. These statements define library features such as technology type, date, and revision, as well as definitions and defaults that apply to the library in general. Every cell in the library has a separate cell description.

For a complete list and the syntax of all the groups and attributes in a technology library, see the *Synopsys Logic Library Reference Manual*.

Example 5-1 General Syntax of the Library Description

```
library (name) {
    technology (name) /* library-level attributes */
    delay_model : table_lookup ;
    bus_naming_style : string ;
    date : string ;
    revision : float_or_string ;
    comment : "string" ;
    time_unit : unit ;
    voltage_unit : unit ;
    current_unit : unit ;
    pulling_resistance_unit : unit ;
    capacitive_load_unit(value,unit);
    leakage_power_unit : unit ;
    define_cell_area(area_name,resource_type) ;
    library_features (value) ;
    /* default environment definitions */
    operating_conditions (name) {
        operating conditions
    }
    wire_load (name) {
        wire load information
    }
    wire_load_selection() {
        area/group selections
    }
    power_lut_template (name) {
```

```
        power lookup table template information
    }
    cell (name1) /* cell definitions */
        cell information
    }
    cell (name2) {
        cell information
    }
    ...
    type (name) {
        bus type name
    }
    input_voltage (name) {
        input voltage information
    }
    output_voltage (name) {
        output voltage information
    }
```

library Group

The `library` group statement defines the name of the library you want to describe. This statement must be the first executable line in your library.

Example

```
library (my_library) {
...
}
```

Using General Library Attributes

These attributes apply generally to the technology library:

- `technology`
- `delay_model`
- `bus_naming_style`

technology Attribute

This attribute identifies the technology used in the library. Valid value is `cmos`. The `technology` attribute must be the first attribute defined and is placed at the top of the listing.

If you do not specify the `technology` attribute, the default is `cmos`.

Syntax

```
technology (name) ;
```

Example

```
library (my_library) {  
    technology (cmos);  
    ...  
}
```

delay_model Attribute

This attribute indicates the delay model to use in delay calculations. Valid value is `table_lookup`.

The `delay_model` attribute must follow the `technology` attribute; if the `technology` attribute is not present, the `delay_model` attribute must be the first attribute in the library.

Syntax

```
delay_model : value ;
```

Example

```
library (my_library) {  
    delay_model : table_lookup;  
    ...  
}
```

bus_naming_style Attribute

This attribute defines the naming convention for buses in the library.

Syntax

```
bus_naming_style : "string";
```

string

Contains alphanumeric characters, braces, underscores, dashes, or parentheses. Must contain one `%s` symbol and one `%d` symbol. The `%s` and `%d` symbols can appear in any order with at least one nonnumeric character in between.

The colon character is not allowed in a `bus_naming_style` attribute value because the colon is used to denote a range of bus members. You construct a complete bused-pin name by using the name of the owning bus and the member number. The owning bus name is substituted for the `%s`, and the member number replaces the `%d`.

If you do not define the `bus_naming_style` attribute, Library Compiler applies the default naming convention, as shown in the following example.

Example

```
bus_naming_style : "Bus%$Pin%d";
```

Note:

You cannot redefine the `bus_naming_style` attribute of a technology library from the `lc_shell` prompt.

Documenting Libraries

Use these library-level attributes to document the library:

- date
 - revision
 - comment
-

date Attribute

This attribute identifies the date your library was created.

Example

```
date : "September 1, 2015";
```

The library report produced by the `report_lib` command shows the date.

revision Attribute

This attribute defines a revision number for your library.

Example

```
revision : 2015.01;
```

comment Attribute

Use this attribute to include information you want printed in the `report_lib` report, such as copyright or other product information. You can include only one comment line in a library. You can have an unlimited number of characters in the string, but the string must be enclosed in quotation marks.

Example

```
comment : "Copyright 2015, General Silicon, Inc."
```

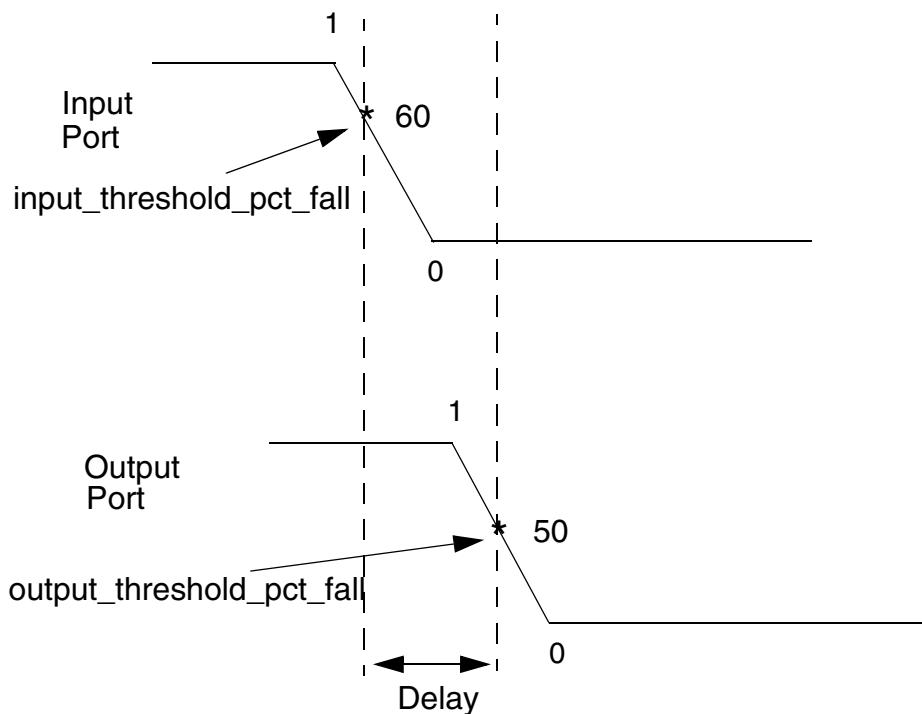
Delay and Slew Attributes

This section describes the attributes to set the values of the input and output pin threshold points that are used to model delay and slew.

Delay is the time it takes for the output signal voltage, which is falling from 1 to 0, to fall to the threshold point set with the `output_threshold_pct_fall` attribute after the input signal voltage, which is falling from 1 to 0, has fallen to the threshold point set with the `input_threshold_pct_fall` attribute (see [Figure 5-1](#)).

Delay is also the time it takes for the output signal, which is rising from 0 to 1, to rise to the threshold point set with the `output_threshold_pct_rise` attribute after the input signal, which is rising from 0 to 1, has risen from 0 to the threshold point set with the `input_threshold_pct_rise` attribute.

Figure 5-1 Delay Modeling for Falling Signal



Slew is the time it takes for the voltage value to fall or rise between two designated threshold points on an input, an output, or a bidirectional port. The designated threshold points must fall within a voltage falling from 1 to 0 or rising from 0 to 1.

Use the following attributes to enter the two designated threshold points to model the time for voltage falling from 1 to 0:

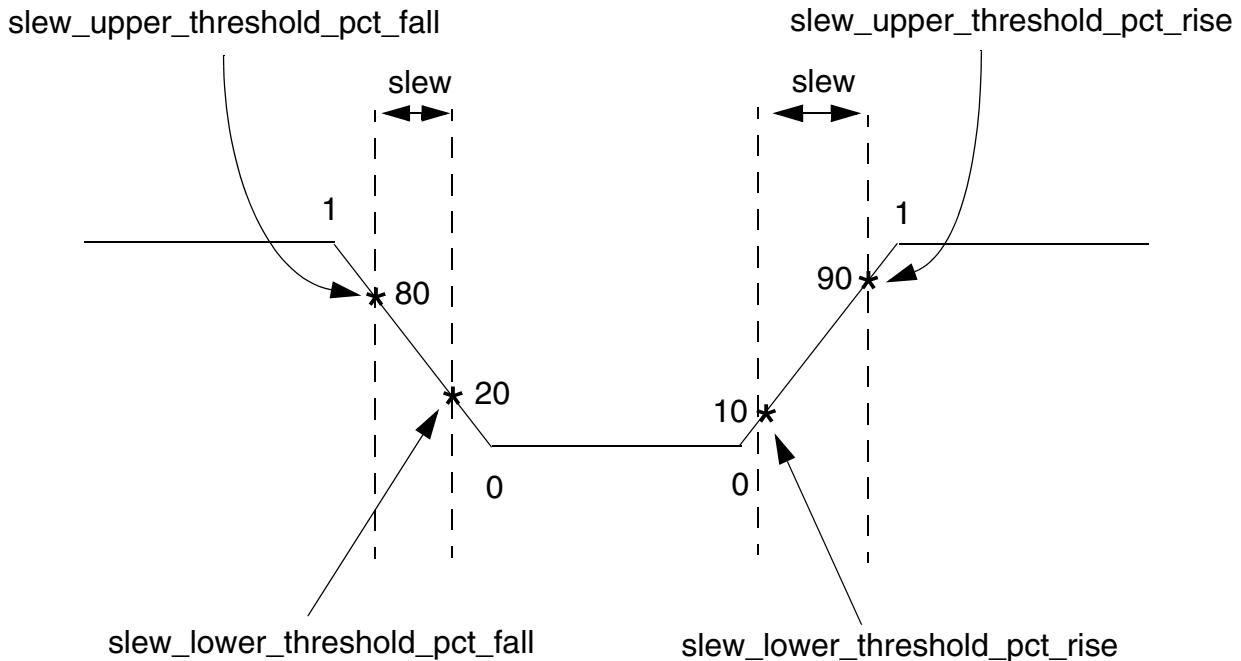
- slew_lower_threshold_pct_fall
- slew_upper_threshold_pct_fall

Use the following attributes to enter the two designated threshold points to model the time for voltage rising from 0 to 1:

- slew_lower_threshold_pct_rise
- slew_upper_threshold_pct_rise

[Figure 5-2](#) shows an example of slew modeling.

Figure 5-2 Slew Modeling



input_threshold_pct_fall Simple Attribute

Note:

To model the delay of a signal going from an input pin to an output pin, you also need to set the value of the output pin. See “[output_threshold_pct_fall Simple Attribute](#)” on page 5-8 and “[output_threshold_pct_rise Simple Attribute](#)” on page 5-9 for details.

Use the `input_threshold_pct_fall` attribute to set the value of the threshold point on an input pin signal falling from 1 to 0. This value is used to model the delay of a signal transmitting from an input pin to an output pin.

Syntax

```
input_threshold_pct_fall : trip_point_value ;  
  
trip_point
```

A floating-point number between 0.0 and 100.0 that specifies the threshold point of an input pin signal falling from 1 to 0. The default is 50.0.

Example

```
input_threshold_pct_fall : 60.0 ;
```

input_threshold_pct_rise Simple Attribute

Use the `input_threshold_pct_rise` attribute to set the value of the threshold point on an input pin signal rising from 0 to 1. This value is used to model the delay of a signal transmitting from an input pin to an output pin.

Note:

To model the delay of a signal going from an input pin to an output pin, you also need to set the value of the output pin. See “[output_threshold_pct_fall Simple Attribute](#)” and “[output_threshold_pct_rise Simple Attribute](#)” on page 5-9 for details.

Syntax

```
input_threshold_pct_rise : trip_point_value ;  
  
trip_point
```

A floating-point number between 0.0 and 100.0 that specifies the threshold point of an input pin signal rising from 0 to 1. The default is 50.0.

Example

```
input_threshold_pct_rise : 40.0 ;
```

output_threshold_pct_fall Simple Attribute

Use the `output_threshold_pct_fall` attribute to set the value of the threshold point on an output pin signal falling from 1 to 0. This value is used to model the delay of a signal transmitting from an input pin to an output pin.

Note:

To model the delay of a signal going from an input pin to an output pin, you also need to set the value of the input pin. See “[input_threshold_pct_rise Simple Attribute](#)” and “[input_threshold_pct_fall Simple Attribute](#)” on page 5-7 for details.

Syntax

```
output_threshold_pct_fall : trip_point_value ;
```

trip_point

A floating-point number between 0.0 and 100.0 that specifies the threshold point of an output pin signal falling from 1 to 0. The default is 50.0.

Example

```
output_threshold_pct_fall : 40.0 ;
```

output_threshold_pct_rise Simple Attribute

Use the `output_threshold_pct_rise` attribute to set the value of the threshold point on an output pin signal rising from 0 to 1. This value is used to model the delay of a signal transmitting from an input pin to an output pin.

Note:

To model the delay of a signal going from an input pin to an output pin, you also need to set the value of the input pin. See “[input_threshold_pct_rise Simple Attribute](#)” on page 5-8 and “[input_threshold_pct_fall Simple Attribute](#)” on page 5-7 for details.

Syntax

```
output_threshold_pct_rise : trip_point_value ;
```

trip_point

A floating-point number between 0.0 and 100.0 that specifies the threshold point of an output pin signal rising from 0 to 1. The default is 50.0.

Example

```
output_threshold_pct_rise : 40.0 ;
```

slew_derate_from_library Simple Attribute

Use the `slew_derate_from_library` attribute to specify how the transition times found in the Synopsys library need to be derated to match the transition times between the characterization trip points.

Syntax

```
slew_derate_from_library : derate_value ;
```

derate

A floating-point number between 0.0 and 1.0. The default is 1.0.

Example

```
slew_derate_from_library : 0.5 ;
```

slew_lower_threshold_pct_fall Simple Attribute

Use the `slew_lower_threshold_pct_fall` attribute to set the value of the lower threshold point that is used to model the delay of a pin falling from 1 to 0.

Note:

To model the delay of a pin falling from 1 to 0, you also need to set the value for the upper threshold point. See “[slew_lower_threshold_pct_rise Simple Attribute](#)” for details.

Syntax

```
slew_lower_threshold_pct_fall : trip_point_value ;
```

trip_point

A floating-point number between 0.0 and 100.0 that specifies the lower threshold point that is used to model the delay of a pin falling from 1 to 0. The default is 20.0.

Example

```
slew_lower_threshold_pct_fall : 30.0 ;
```

slew_lower_threshold_pct_rise Simple Attribute

Use the `slew_lower_threshold_pct_rise` attribute to set the value of the lower threshold point that is used to model the delay of a pin rising from 0 to 1.

Note:

To model the delay of a pin rising from 0 to 1, you also need to set the value for the upper threshold point. See “[slew_upper_threshold_pct_fall Simple Attribute](#)” on page 5-11 for details.

Syntax

```
slew_lower_threshold_pct_rise : trip_point_value ;
```

trip_point

A floating-point number between 0.0 and 100.0 that specifies the lower threshold point that is used to model the delay of a pin rising from 0 to 1. The default is 20.0.

Example

```
slew_lower_threshold_pct_rise : 30.0 ;
```

slew_upper_threshold_pct_fall Simple Attribute

Use the `slew_upper_threshold_pct_fall` attribute to set the value of the upper threshold point that is used to model the delay of a pin falling from 1 to 0.

Note:

To model the delay of a pin falling from 1 to 0, you also need to set the value for the lower threshold point. See “[slew_lower_threshold_pct_fall Simple Attribute](#)” on page 5-10 for details.

Syntax

```
slew_upper_threshold_pct_fall : trip_point_value ;
```

trip_point

A floating-point number between 0.0 and 100.0 that specifies the upper threshold point that is used to model the delay of a pin falling from 1 to 0. The default is 80.0.

Example

```
slew_upper_threshold_pct_fall : 70.0 ;
```

slew_upper_threshold_pct_rise Simple Attribute

Use the `slew_upper_threshold_pct_rise` attribute to set the value of the upper threshold point that is used to model the delay of a pin rising from 0 to 1.

Note:

To model the delay of a pin rising from 0 to 1, you also need to set the value for the lower threshold point. See “[slew_lower_threshold_pct_rise Simple Attribute](#)” on page 5-10 for details.

Syntax

```
slew_upper_threshold_pct_rise : trip_point_value ;
```

trip_point

A floating-point number between 0.0 and 100.0 that specifies the upper threshold point that is used to model the delay of a pin rising from 0 to 1. The default is 80.0.

Example

```
slew_upper_threshold_pct_rise : 70.0 ;
```

Defining Units

Use these library-level attributes to define units:

- `time_unit`
- `voltage_unit`
- `current_unit`
- `pulling_resistance_unit`
- `capacitive_load_unit`
- `leakage_power_unit`

The unit attributes identify the units of measure, such as nanoseconds or picofarads, used in the library definitions.

The Library Compiler tool does not do any conversions.

time_unit Attribute

Use this attribute to identify the physical time unit used in the generated library.

Syntax

`time_unit : unit ;`

unit

Valid values are 1ps, 10ps, 100ps, and 1ns. The default is 1ns.

Example

`time_unit : "10ps";`

voltage_unit Attribute

Use this attribute to scale the contents of the `input_voltage` and `output_voltage` groups. Additionally, the `voltage` attribute in the `operating_conditions` group represents values in the voltage units. See [Chapter 6, “Building Environments”](#) for more information.

Syntax

`voltage_unit : unit ;`

unit

Valid values are 1mV, 10mV, 100mV, and 1V. The default is 1V.

Example

```
voltage_unit : "100mV";
```

current_unit Attribute

This attribute specifies the unit for the drive current that is generated by output pads. The `pulling_current` attribute for a pull-up or pull-down transistor also represents its values in this unit.

Syntax

```
current_unit : value_enum ;
```

value

The valid values are 1uA, 10uA, 100uA, 1mA, 10mA, 100mA, and 1A. No default exists for the `current_unit` attribute if the attribute is omitted.

Example

```
current_unit : "1mA";
```

pulling_resistance_unit Attribute

The `pulling_resistance_unit` attribute defines pulling resistance unit values for pull-up and pull-down devices.

Syntax

```
pulling_resistance_unit : "unit" ;
```

unit

Valid unit values are 1ohm, 10ohm, 100ohm, and 1kohm. No default exists for `pulling_resistance_unit` if the attribute is omitted.

Example

```
pulling_resistance_unit : "10ohm";
```

capacitive_load_unit Attribute

This attribute specifies the unit for all capacitance values within the technology library, including default capacitances, max_fanout capacitances, pin capacitances, and wire capacitances.

Syntax

```
capacitive_load_unit (valuefloat,unitenum) ;
```

value

A floating-point number.

unit

Valid values are ff and pf.

Example

```
capacitive_load_unit(1,pf);
```

leakage_power_unit Attribute

This attribute indicates the units of the power values in the library. If this attribute is missing, the leakage-power values are expressed without units.

Syntax

```
leakage_power_unit : valueenum ;
```

value

Valid values are 1mW, 100mW, 10mW, 1mW, 100nW, 10nW, 1nW, 100pW, 10pW, and 1pW.

Example

```
leakage_power_unit : 100uW;
```

Describing Pads

Use these library-level groups and attributes for pad descriptions:

- input_voltage group
- output_voltage group

input_voltage Group

This group captures a set of voltage levels at which an input pad is driven.

Note:

You can also use the `input_voltage` group to specify voltages ranges for standard cells.

Example

```
library (my_library) {  
...  
    input_voltage(CMOS) {  
        vil : 0.3 * VDD;  
        vih : 0.7 * VDD;  
        vimin : -0.5;  
        vimax : VDD + 0.5;  
    }  
}
```

You can use the `input_voltage` and `output_voltage` groups to define a set of input or output voltage ranges for your pads. You can then assign this set of voltage ranges to the input or output pin of a pad cell.

For example, you can define an `input_voltage` group called TTL with a set of high and low thresholds and minimum and maximum voltage levels. Use the following command in the `pin` group to assign those ranges to the pad cell pin.

```
input_voltage : TTL ;
```

The defaults represent nominal operating conditions. These values fluctuate with the voltage range defined in the operating conditions groups.

All voltage values are in the units you define with the `library group voltage_unit` attribute.

output_voltage Group

This group captures a set of voltage levels at which an output pad is driven.

Note:

You can also use the `input_voltage` group to specify voltages ranges for standard cells.

The defaults represent nominal operating conditions. These values fluctuate with the voltage range defined in the operating conditions groups.

All voltage values are in the units you define with the `library group voltage_unit` attribute.

Example

```
library (my_library) {
...
    output_voltage(GENERAL) {
        vol : 0.4;
        voh : 2.4;
        vomin : -0.e;
        vomax : VDD + 0.3;
    }
}
```

Describing Power

You can describe power dissipation in libraries, using the CMOS nonlinear delay model.

To describe power,

1. Use the library-level `power_lut_template` group to define templates of common information to use with lookup tables.
2. Use the template and the cell-level `internal_power` group in the `pin` group to create lookup tables of power information (see [Chapter 12, “Modeling Power and Electromigration”](#)).

Lookup tables and their corresponding templates can be one-, two-, or three-dimensional.

`power_lut_template` Group

Use this library-level group to create templates of common information that multiple lookup tables can use. A table template specifies the table parameters and the breakpoints for each axis. Assign each template a name. So that power lookup tables can refer to the template, place its name as the group name of a `fall_power` group, `power` group, or `rise_power` group in the `internal_power` group (see the next section, “[Template Variables](#)”).

The syntax for the `power_lut_template` group is

```
power_lut_template(name) {
    variable_1 : string ;
    variable_2 : string ;
    variable_3 : string ;
    index_1("float, ... , float") ;
    index_2("float, ... , float") ;
    index_3("float, ... , float") ;
}
```

Template Variables

The template for specifying power uses three associated variables to characterize cells in the library for internal power:

- `variable_1`, which specifies the first-dimensional variable
- `variable_2`, which specifies the second-dimensional variable
- `variable_3`, which specifies the third-dimensional variable

These variables indicate the parameters used to index into the lookup table along the first, second, and third table axes.

Following are valid values for `variable_1`, `variable_2`, and `variable_3`:

`total_output_net_capacitance`

This is the loading of the output net capacitance of the pin specified in the `pin` group that contains the `internal_power` group.

`equal_or_opposite_output_net_capacitance`

This is the loading of the output net capacitance of the pin specified in the `equal_or_opposite_output` attribute in the `internal_power` group.

`input_transition_time`

This is the input transition time of the pin specified in the `related_pin` attribute in the `internal_power` group.

For information about the `related_pin` attribute, see [Chapter 11, “Timing Arcs”](#).

Template Breakpoints

The following index statements define the breakpoints for an axis.

- `index_1` specifies the breakpoints of the first dimension defined by `variable_1`.
- `index_2` specifies the breakpoints of the second dimension defined by `variable_2`.
- `index_3` specifies the breakpoints of the third dimension defined by `variable_3`.

You can overwrite the `index_1`, `index_2`, and `index_3` attribute values by providing the same `index_x` attributes in the `fall_power` group, `power` group, or `rise_power` group in the `internal_power` group.

The index values are lists of floating-point numbers greater than or equal to 0.0. The values in the list must be in increasing order.

The number of floating-point numbers in the indexes determines the size of each dimension, as [Example 5-2](#) illustrates.

You must define at least one `index_1` statement in the `power_lut_template` group. For a one-dimensional table, use only `variable_1`.

[Example 5-2](#) shows four `power_lut_template` groups that have one-, two-, and three-dimensional templates.

Example 5-2 Four power_lut_template Groups

```
power_lut_template (output_by_cap) {
    variable_1 : total_output_net_capacitance ;
    index_1 ("0.0, 5.0, 20.0") ;
}

power_lut_template (output_by_cap_and_trans) {
    variable_1 : total_output_net_capacitance ;
    variable_2 : input_transition_time ;
    index_1 ("0.0, 5.0, 20.0") ;
    index_2 ("0.1, 1.0, 5.0") ;
}

power_lut_template (input_by_trans) {
    variable_1 : input_transition_time ;
    index_1 ("0.0, 1.0, 5.0") ;
}

power_lut_template (output_by_cap2_and_trans) {
    variable_1 : total_output_net_capacitance ;
    variable_2 : input_transition_time ;
    variable_3 : total_equal_or_opposite_net_cap ;
    index_1 ("0.0, 5.0, 20.0") ;
    index_2 ("0.1, 1.0, 5.0") ;
    index_3 ("0.1, 0.5, 1.0") ;
}
```

Note:

There is a predefined template whose name is `scalar` and whose size is 1. You can refer to it by placing the string, `scalar`, as the group name of a `fall_power` group, `power_group`, or `rise_power` group in the `internal_power` group.

Using Processing Attributes

Use the `library_features` attribute to define processing options.

The `library_features` attribute lets other Synopsys products use the command features that you specify as attribute values.

The default for the `library_features` attribute is none (no library features are available for use by other Synopsys products).

Syntax

```
library_features (value) ;
```

`value`

Valid values are `report_delay_calculation`, `report_power_calculation`, `report_noise_calculation`, `report_user_data` and `allow_update_attribute`. The default is none (no library features are available to be used by other Synopsys products).

Example

```
library_features (report_delay_calculation) ;
```

Setting Minimum Cell Requirements

For the Design Compiler tool to perform technology mapping and optimization on design descriptions, the technology library should contain a minimum set of cells.

This is a minimum set of cells for a CMOS technology library:

- An inverter
- A 2-input NAND gate
- A 2-input NOR gate
- A three-state buffer
- A D flip-flop with preset, clear, and complementary output values
- A D latch with preset, clear, and complementary output values

The Design Compiler tool writes out a warning message if the design contains a cell that is not included in the library or libraries associated with the design.

Setting CMOS Default Attributes

You can set default pin and timing attribute values for a CMOS technology library in the `library` group definition. You can override defaults with attribute values set at the individual `pin` or `timing` group level.

Table 5-1 lists the default attributes you can use in the CMOS nonlinear delay models. The table lists the default attributes that you can define within the `library` group and the attributes that override them.

For descriptions of the default attributes, see [Chapter 6, “Building Environments”](#).

Table 5-1 CMOS Default Attributes

Default attribute	Description	Override with
<code>default_cell_leakage_power</code>	Default leakage power	<code>cell_leakage_power</code>
<code>default_connection_class</code>	Default connection class	<code>connection_class</code>
<code>default_fanout_load</code>	Fanout load of input pins	<code>fanout_load</code>
<code>default inout pin cap</code>	Capacitance of inout pins	<code>capacitance</code>
<code>default input pin cap</code>	Capacitance of input pins	<code>capacitance</code>
<code>default_max_capacitance</code>	Maximum capacitance of output pins	<code>max_capacitance</code>
<code>default_max_fanout</code>	Maximum fanout of all output pins	<code>max_fanout</code>
<code>default_max_transition</code>	Maximum transition of output pins	<code>max_transition</code>
<code>default_operating_conditions</code>	Default operating conditions for the library	<code>operating_conditions</code>
<code>default output pin cap</code>	Capacitance of output pins	<code>capacitance</code>
<code>default wire load</code>	Wire load	No override available
<code>default wire load area</code>	Wire load area	No override available
<code>default wire load capacitance</code>	Wire load capacitance	No override available

Table 5-1 CMOS Default Attributes (Continued)

Default attribute	Description	Override with
default_wire_load_mode	Wire load mode	set_wire_load -mode in dc_shell
default_wire_load_resistance	Wire load resistance	No override available
default_wire_load_selection	Wire load selection	No override available

6

Building Environments

Variations in operating temperature, supply voltage, and manufacturing process cause performance variations in electronic networks.

The Library Compiler tool gives you the ability to model these variations. Then the Design Compiler tool can use the models to modify the synthesis and optimization environment.

To create these models, you use environment attribute statements in a technology library. All environment attributes have built-in default settings for typical delays. If you run Design Compiler without variables, the optimization process uses the default delays. Alternatively, you can create your own default settings at the library level.

The environment attributes include various attributes and tasks, covered in the following sections:

- [Library-Level Default Attributes](#)
- [Defining Operating Conditions](#)
- [Defining Wire Load Groups](#)
- [Selecting Wire Load Groups Automatically](#)
- [Specifying Delay Scaling Attributes](#)

Library-Level Default Attributes

Global defaults are set at the library level. You can override many of these defaults.

Setting Default Cell Attributes

The following attributes are defaults that apply to all cells in a library.

default_cell_leakage_power Simple Attribute

Indicates the default leakage power for those cells that do not have the `cell_leakage_power` attribute. This attribute must be a nonnegative floating-point number. If it is not defined, this attribute defaults to 0.0.

Example

```
default_cell_leakage_power : 0.5;
```

The Library Compiler tool issues a warning if the library has `cell_leakage_power` information but does not have the `default_cell_leakage_power` attribute defined.

Setting Default Pin Attributes

Default pin attributes apply to all pins in a library and deal with timing. How you define default timing attributes in your library depends on the timing delay model you use.

These are the defaults that apply to all pins in a library.

```
default inout pin cap : value float ;
```

Sets a default for `capacitance` for all I/O pins in the library.

```
default input pin cap : value float ;
```

Sets a default for `capacitance` for all input pins in the library.

```
default output pin cap : value float ;
```

Sets a default for `capacitance` for all output pins in the library.

```
default max fanout : value float ;
```

Sets a default for `max_fanout` for all output pins in the library.

```
default max transition : value float ;
```

Sets a default for `max_transition` for all output pins in the library.

```
default_fanout_load : value_float ;
```

Sets a default for `fanout_load` for all input pins in the library.

The following example shows the default pin attributes in a CMOS library:

Example 6-1 Default Pin Attributes for a CMOS Library

```
library (example) {
  ...
  /* default pin attributes */
  default_inout_pin_cap      : 1.0 ;
  default_input_pin_cap       : 1.0 ;
  default_output_pin_cap      : 0.0 ;
  default_fanout_load        : 1.0 ;
  default_max_fanout          : 10.0 ;
  default_max_transition      : 15.0 ;
  ...
}
```

Setting Wire Load Defaults

Use the following library-level attributes to set wire load defaults.

default_wire_load Attribute

Assigns the defaults to the `wire_load` group, unless you assign a different value for `wire_load` before compiling the design.

You can select a `wire_load` automatically by using `wire_load_selection`. This allows Design Compiler to select a `wire_load` group automatically and use it for wire load estimation.

Syntax

```
default_wire_load : wire_load_name;
```

Example

```
default_wire_load : WL1;
```

Synchronizing Design and Model Modes

The calculations the synthesis tool uses to develop net delays depend, in part, on the design's wire load mode. When you attach a wire load mode to a design, that new mode must match the sampling mode that the semiconductor vendor used when creating the design's wire load model.

default_wire_load_capacitance Attribute

Specifies a value for the default wire load capacitance.

Syntax

```
default_wire_load_capacitance : value;
```

Example

```
default_wire_load_capacitance : .05;
```

default_wire_load_resistance Attribute

Specifies a value for the default wire load resistance.

Syntax

```
default_wire_load_resistance : value;
```

Example

```
default_wire_load_resistance : .067;
```

default_wire_load_area Attribute

Specifies a value for the default wire load area.

Syntax

```
default_wire_load_area : value;
```

Example

```
default_wire_load_area : 0.33;
```

Setting Other Environment Defaults

Use the following library-level attributes to set other environment defaults.

default_operating_conditions Attribute

Assigns a `default operating_conditions` group name for the library. It must be specified after all `operating_conditions` groups. If this attribute is not used, nominal operating conditions apply. See “[Defining Operating Conditions](#)” on page 6-6.

Syntax

```
default_operating_conditions : operating_condition_name;
```

Example

```
default_operating_conditions : WCCOM1;
```

default_connection_class Attribute

Sets a default for `connection_class` for all pins in a library.

Example

```
default_connection_class : name1 [name2 name3 ...];
```

Examples of Library-Level Default Attributes

Example 6-2 illustrates a library-level default attribute setting for a CMOS library. The `wire_load` and `operating_conditions` group statements illustrate the requirement that group names that are referred to by the default attributes, such as `WL1` and `OP1`, must be defined in the library.

Example 6-2 Setting Library-Level Defaults for a CMOS Library

```
library (example) {
    ...
    /* default cell attributes */

    default_cell_leakage_power : 0.2;

    /* default pin attributes */

    default inout_pin_cap : 1.0;
    default input_pin_cap : 1.0;
    default output_pin_cap : 0.0;
    default fanout_load : 1.0;
    default max_fanout : 10.0;

    wire_load (WL1) {
        ...
    }
    operating_conditions (OP1) {
        ...
    }
    default wire_load : WL1;
    default operating_conditions : OP1;
    default wire_load_mode : enclosed;
    ...
}
```

em_temp_degradation_factor Attribute

Specifies the electromigration temperature exponential degradation factor. The factor is specified as a floating-point number.

If this optional attribute is not defined, the nominal temperature electromigration tables are used for all operating temperatures.

Syntax

```
em_temp_degradation_factor : "float" ;
```

float

A floating-point number in centigrade units consistent with other temperature specifications throughout the library.

Example

```
em_temp_degradation_factor : 40.0 ;
```

Defining Operating Conditions

The following section explains how to define and determine various operating conditions for a technology library.

operating_conditions Group

An `operating_conditions` group is defined in a library group.

Syntax

```
library ( lib_name ) {  
    operating_conditions ( name ) {  
        ... operating  
        conditions description ...  
    }  
}
```

name

Identifies the set of operating conditions. Names of all `operating_conditions` groups and `wire_load` groups must be unique within a library.

The `operating_conditions` groups are useful for testing timing and other characteristics of your design in predefined simulated environments. The following attributes are defined in an `operating_conditions` group:

```
process : multiplier ;
```

The scaling factor accounts for variations in the outcome of the actual semiconductor manufacturing steps, typically 1.0 for most technologies. The multiplier is a floating-point number from 0 through 100.

```
process_label : "name" ;
```

The process name of the current process. The value is a string.

```
temperature : value ;
```

The ambient temperature in which the design is to operate. The value is a floating-point number.

```
voltage : value ;
```

The operating voltage of the design, typically 5 volts for a CMOS library. The value is a floating-point number from 0 through 1,000, representing the absolute value of the actual voltage.

Note:

Define voltage units consistently.

See the section on the `voltage_unit` attribute in [Chapter 5, “Building a Technology Library”](#).

```
tree_type : model ;
```

The definition for the environment interconnect model.

The Design Compiler tool uses the interconnect model to select the formula for calculating interconnect delays.

The model is one of the following three models:

- *best_case_tree*

Models the case in which the load pin is physically adjacent to the driver. In the best case, all wire capacitance is incurred but none of the wire resistance must be overcome.

- *balanced_tree*

Models the case in which all load pins are on separate, equal branches of the interconnect wire. In the balanced case, each load pin incurs an equal portion of the total wire capacitance and wire resistance.

- *worst_case_tree*

Models the case in which the load pin is at the extreme end of the wire. In the worst case, each load pin incurs both the full wire capacitance and the full wire resistance.

Defining Timing Ranges at Runtime

You can define one or two timing ranges in dc_shell to use when optimizing a design in Design Compiler. The `set_timing_ranges` command defines the timing ranges used in the design.

Syntax

```
set_timing_ranges ( range_name [, range_name2] )
```

Note:

This modeling method produces accurate results in cases where all the delays have a linear dependency on the operating parameters. In cases where delays do not scale linearly with operating conditions, this method provides only a first-order approximation.

Timing ranges influence constraint evaluation only; they do not affect timing reports in Design Compiler. You can obtain information about timing ranges that are defined in a library by using the `report_lib` command.

Following is an example of the timing-range information from the `set_timing_ranges` command. This example illustrates two ranges, `FAST_RANGE` and `SLOW_RANGE`, with their respective limits.

Timing Ranges:

Name	Library	Slower factor	Faster factor
FAST_RANGE	test	0.9600	0.9000
SLOW_RANGE	test	1.2000	1.0500

Defining Wire Load Groups

Use the `wire_load` group and the `wire_load_selection` group to specify values for the capacitance factor, resistance factor, area factor, slope, and fanout_length you want to apply to the wire delay model for different sizes of circuitry.

wire_load Group

The `wire_load` group has an extended `fanout_length` complex attribute. Define the `wire_load` group at the library level.

Syntax

```
wire_load(name) {
  resistance : value ;
  capacitance : value ;
  area : value ;
```

```
slope : value ;
fanout_length(fanout_int, length_float, \
              average_capacitance_float, standard_deviation_float, \
              number_of_nets_int);
}
```

A `wire_load` group contains all the information Design Compiler needs to estimate interconnect wiring delays.

In a `wire_load` group, you define the estimated wire length as a function of fanout. You can also define scaling factors to derive wire resistance, capacitance, and area from a given length of wire.

You can define any number of `wire_load` groups in a technology library, but all `wire_load` groups and `operating_conditions` groups must have unique names.

You can define the following simple attributes in a `wire_load` group:

`resistance : value ;`

Specifies a floating-point number representing wire resistance per unit length of interconnect wire.

`capacitance : value ;`

Specifies a floating-point number representing capacitance per unit length of interconnect wire.

`area : value ;`

Specifies a floating-point number representing the area per unit length of interconnect wire.

`slope : value ;`

Specifies a floating-point number representing slope. This attribute characterizes linear fanout length behavior beyond the scope of the longest length described by the `fanout_length` attributes.

You can define the following complex attribute in a `wire_load` group:

```
fanout_length ( fanout_int, length_float, average_capacitance_float \
                standard_deviation_float , number_of_nets_int );
```

`fanout_length` is a complex attribute that defines values that represent fanout and length. The `fanout` value is an integer; `length` is a floating-point number.

When you create a wire load manually, define only `fanout` and `length`.

You must define at least one pair of `fanout` and `length` points per wire load model. You can define as many additional pairs as necessary to characterize the fanout-length behavior you want.

When the Design Compiler tool encounters multiple *fanout* and *length* points, it uses linear interpolation to determine points between them, and extrapolation beyond the last point.

```
interconnect_delay (template_name)
{values(float,...float,...float,...float,...) ; }
```

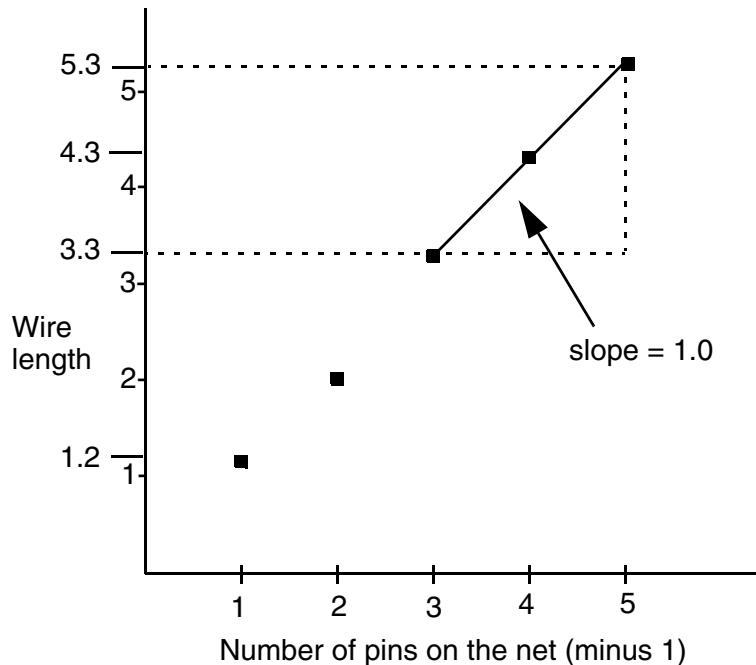
The `interconnect_delay` group specifies the lookup table template and the wire delay values.

Specify the `interconnect_delay` values.

To overwrite the default index values, specify the new index values before the `interconnect_delay` values, as shown here.

[Figure 6-1](#) illustrates the correlation between the number of pins on the net, excluding the driver pin, and the estimated wire length of the metal between all the pins.

Figure 6-1 Wire Load Definition Graph



[Example 6-3](#) gives the library description for the model in [Figure 6-1](#).

Note:

In [Example 6-3](#) and [Example 6-4](#), the name 90x90 is enclosed in quotation marks because it is a string that begins with a number.

Example 6-3 Wire Load Definition

```
library (example) {
```

```

...
    wire_load ("90x90") {
        resistance : 0;
        capacitance : 1;
        area : 0;
        slope : 1.0;
        fanout_length( 1, 1.2 );
        fanout_length( 2, 2.0 );
        fanout_length( 3, 3.3 );
    }
}

```

[Example 6-4](#) shows how to estimate the wire delay by using a 3-D lookup table.

Example 6-4 Wire Delay Estimation, Using 3-D Lookup Table

```

library (example) {
...
lu_table_template (wire_delay_table_template) {
    variable_1 : fanout_number;
    variable_2 : fanout_pin_capacitance;
    variable_3 : driver_slew;
    index_1( "0.12,3.4" );
    index_2( "0.12,4.24" );
    index_3( "0.1,2.7,3.12" );
}
}

```

You can define one `wire_load` group per design at Design Compiler runtime. This is the syntax for the `set_wire_load` command in `dc_shell`:

```
set_wire_load wire_load_name [-library] [-mode]
```

The `wire_load_name` is the name of a `wire_load` group defined in the library. To use the `wire_load` group defined in [Example 6-3](#), enter

```
dc_shell> set_wire_load "90x90"
```

For more information, see the `set_wire_load` command in the Synopsys man pages.

If you don't define a `wire_load` group, the Library Compiler tool uses the `default_wire_load_resistance` and `default_wire_load_capacitance` attribute values, respectively, for wire resistance and wire capacitance.

The `report_lib` command reports the information for `default_wire_load_capacitance`, `default_wire_load_resistance`, and `default_wire_load_area`. The additional wire load information is reported in the Library Compiler report if it is available.

[Example 6-5](#) shows the report for a wire load model containing new `fanout` and `length` information.

Example 6-5 Wire Load Model Containing New Fanout Length Information

```
Wire Loading Model:
Name      : 05x05
Location   : wl library_name
Resistance : 0
Capacitance: 1
Area       : 0
Slope      : 0.186

Fanout    Length     Points      AverageCap      StdDeviation
-----  -----  -----
1         0.39      50          1.30            0.02
```

Calculating Wire Area

The Library Compiler tool calculates wire area by using one of three methods:

- Using values specified when the fanout values correspond to one of the values specified in the model
- Extrapolation
- Interpolation

[Example 6-6](#) shows a wire load model.

Example 6-6 Wire Load Model

```
wire_load(standard) {
    resistance : 0.001;
    capacitance : 1.2 ;
    area : 0.5;
    slope : 0.311 ;
    fanout_length(1,0.53);
    fanout_length(2,0.63);
    fanout_length(5,0.83);
}
```

Using Values Specified

In the example, the area value that is specified in the wire load model is the area per unit length of wire. The `fanout_length` is an estimation of the wire length for a given number of fanouts. So if the driving cell has a fanout of 2, area is calculated as

$$\begin{aligned} \text{Net area} &= \text{area/unit_length} \times \text{fanout_length}(\text{for given fanout}) \\ &= 0.5 \times 0.63 = 0.315 \end{aligned}$$

Extrapolation

For fanout greater than the largest fanout in the table, the Library Compiler tool uses the slope (specified in the wire load model) and the largest fanout values. In the example, the fanout is 6, so area is calculated as

```
Net area[n] = Net area[last fanout] + ((fanout-last fanout) x
                                         slope x area/unit_length)
Net area[6] = {(Net area[5]) + ( (6-5) x 0.311 x 0.5 ) }
Net area[5] = Area/unit_length X fanout_length (for fanout of 5)
             = 0.5 x 0.83 = 0.42
```

Therefore, the net area for fanout 6 is

$$\text{Net area}[6] = \{(0.42) + (1 \times 0.311 \times 0.5)\} = 0.5755$$

For other fanout values, the calculation is

$$\begin{aligned}\text{Net area}[7] &= \{(0.42) + (2 \times 0.311 \times 0.5)\} \\ \text{Net area}[8] &= \{(0.42) + (3 \times 0.311 \times 0.5)\}\end{aligned}$$

or

$$\text{Net area}[7] = \{(0.5755) + (1 \times 0.311 \times 0.5)\}$$

Interpolation

When values are missing from the model, the Library Compiler tool uses interpolation. For a fanout of 3, there is no value in the model, so the calculation is

$$\begin{aligned}\text{Slope}[between 2 and 5] &= (\text{length}[5]-\text{length}[2]) / (5-2) \\ &= (0.83-0.63) / (5-2) = 0.0666 \\ \text{Net area}[2] &= 0.5 \times 0.63 = 0.32 \\ \text{Net area}[n] &= \text{Net area}[last fanout] + ((fanout-last fanout) x \\ &\quad \text{slope} x \text{area/unit_length}) \\ \text{Net area}[3] &= \text{Net area}[2] + ((3-2) x \text{slope}[between 2 and 5] x \\ &\quad \text{area/unit_length}) \\ \text{Net area}[3] &= 0.32 + (1 \times 0.0666 \times 0.5) = 0.3533\end{aligned}$$

The two fanouts closest to 3 are 2 and 5. The Library Compiler tool calculates the slope between fanout 2 and 5.

For a net area of 4, the calculation is

$$\text{Net area}[4] = \{ 0.32 + (2 \times 0.0666 \times 0.5) \} = 0.3866$$

or

$$\text{Net area}[4] = \{ 0.3533 + (1 \times 0.0666 \times 0.5) \} = 0.3866$$

wire_load_table Group

You can use the `wire_load_table` group to estimate accurate connect delay. Compared to the `wire_load` group, this group is more flexible, because wire capacitance and resistance no longer have to be strictly proportional to each other. In some cases, this results in more-accurate connect delay estimates.

Syntax

```
wire_load_table(name_string) {
    fanout_length(fanout_int, length_float);
    fanout_capacitance(fanout_int, capacitance_float);
    fanout_resistance(fanout_int, resistance_float);
    fanout_area(fanout_int, area_float);
}
```

In the `wire_load` group, the `fanout_capacitance`, `fanout_resistance`, and `fanout_area` values represent per-length coefficients. In the `wire_load_table` group the values are exact.

The `report_lib` command reports the `wire_load_table` group. [Example 6-7](#) shows a source file containing a `wire_load_table` group and the corresponding report.

Example 6-7 Wire_load_table With the Table Report

```
library(wlut) {
    wire_load_table("05x05") {
        fanout_length(1, 0.2) ;
        fanout_capacitance(1, 0.15);
        fanout_resistance(1, 0.17) ;
        fanout_area(1, 0.2) ;
        fanout_length(2, 0.35) ;
        fanout_capacitance(2, 0.39) ;
        fanout_resistance(2, 0.25) ;
        fanout_area(2, 0.41) ;
    }
}

Name      : 05x05
Location   : wlut
Fanout     Length      Capacitance      Resistance      Area
-----
1          0.2          0.15            0.17            0.2
2          0.35         0.39            0.25            0.41
```

Selecting Wire Load Groups Automatically

The `wire_load_selection` groups that are defined at the library group level let Design Compiler select a `wire_load` group for wire load estimation automatically. Selection is based on the total cell area of the design.

You use the `selection_group` option of the `set_wire_load` command to determine the `wire_load_selection` group. If no group is selected, the `default_wire_load_selection` attribute determines the default `wire_load_selection` group.

Typically, definitions for `wire_load` groups are built according to statistical experiments with blocks of different sizes. The `wire_load_selection` group incorporates this information in the technology library. Design Compiler uses the information during compilation to achieve better wiring-area and delay-estimation accuracy.

Use multiple selection groups (see [Example 6-8](#)) to allow for different amounts of wire load pessimism or in cases where the same library is used for different fabrication processes.

Example 6-8 Specification of Multiple wire_load_selection Groups

```
wire_load_selection(really_pessimistic) {
    wire_load_from_area(min_area1,max_area1,wire_load_name1);
    wire_load_from_area(min_area2,max_area2,wire_load_name2);
}

wire_load_selection(somewhat_pessimistic) {
    wire_load_from_area(min_area3,max_area3,wire_load_name3);
    wire_load_from_area(min_area4,max_area4,wire_load_name4);
}
default_wire_load_selection : somewhat_pessimistic;
```

With multiple `wire_load_selection` groups, you can put two-layer as well as three-layer wire load models into a single library. You must include the `default_wire_load_selection` attribute to determine the default `wire_load_selection` group.

Within a given `wire_load_selection` group, use `wire_load_from_area` to associate each `wire_load` group with a different area range. This practice is useful for compiling different levels of the design hierarchy separately.

wire_load_from_area Attribute

Use `wire_load_from_area` to associate each `wire_load` group with a different area range. This practice is useful for compiling different levels of the design hierarchy separately.

This attribute must be specified after all `wire_load` groups.

Syntax

```
wire_load_selection(name1) {
    wire_load_from_area(min_areal,max_areal,Wire_load_name1);
    min_areal2,max_areal2,Wire_load_name2
}
wire_load_selection(name-n) {
...
}
```

The `min_area` and `max_area` values give the area range in library cell area units. This range must not overlap a range defined in another `wire_load_from_area` line. The `wire_load_name` is the name of the `wire_load` group to use when the area of the design falls within the defined range, as shown in [Example 6-9](#). A design with an area outside the listed range is assigned the `wire_load` group of the next area range. In [Example 6-9](#), a design with an area of 120 gets the `10x10` `wire_load` group.

[Example 6-9](#) Wire Load Group Selection

```
wire_load_selection(name1) {
    wire_load_from_area(0,100,"05x05");
    wire_load_from_area(150,200,"10x10");
}
```

In [Example 6-10](#), a design with an area of 20 gets the `05x05` `wire_load` group.

[Example 6-10](#) Wire Load Group Selection

```
wire_load_selection(name1)
    wire_load_from_area(50,100,"05x05");
}
```

Specifying Default Wire Load Settings

It is practical to set the `default_wire_load_mode` to enclosed or segmented instead of top. If the `default_wire_load_mode` is set to top, all nets in both the top design and subblocks use the wire load model selected from the area of the top design. If the `default_wire_load_mode` is set to enclosed, the nets fully enclosed within a design use the wire load model selected from the area of that subdesign. If the `default_wire_load_mode` is set to segmented, the nets partially enclosed within a design use the wire load model selected from the area of that subdesign.

Although the previous mechanism for defining a single `wire_load_selection` group is available, it is the exception. Use the `default_wire_load_selection` attribute at the library level when you want to specify more than one group:

```
default_wire_load_selection : name1;
```

Until the design is completely mapped, the total area is unknown. Therefore, Design Compiler uses the `default_wire_load` value, if defined, or none if not defined. Design

Compiler selects the wire load model to use before and during timing operations and operations that modify the design, such as update_timing, report_timing, compile, and translate.

You can override the Design Compiler `wire_load_model` selection with the `set_wire_load` command. You can turn automatic selection of `wire_load_model` off when you are working in `dc_shell`, by setting the `auto_wire_load_selection` to false. For example, in `dc_shell`, enter

```
dc_shell> set auto_wire_load_selection false
```

You can also turn automatic selection of the `wire_load_model` off by modifying the `.synopsys_dc.setup` file. Set the `auto_wire_load_selection` to false.

See the Synopsys man pages for more information about `auto_wire_load_selection`.

The `wire_load_model` the Design Compiler tool selects is reported by the `report_lib` command, which lists available wire loads in the library. The compile log reports changes in the wire load model with an informational message.

Specifying Delay Scaling Attributes

The Design Compiler tool calculates delay estimates by using the scaling factors set in the technology library environment.

These k-factors (attributes that begin with `k_`) are multipliers that scale defined library values, taking into consideration the effects of changes in process, temperature, and voltage.

To model the effects of process, temperature, and voltage variations on circuit timing, use the following:

- k-factors that apply to the entire library and are defined at the library level.
The Library Compiler tool assigns a value of 0 (zero) to any k-factors not defined in the library.
- User-selected operating conditions that override the values in the library for an individual cell.

Using these values, downstream tools uniformly scale timing numbers for timing analysis.

Calculating Delay Factors

Any delay equation factor not affected by process, temperature, or voltage must have the corresponding k-factor set to 0. The default for any unspecified k-factor is 0.

Calculating Voltage Delay Factors

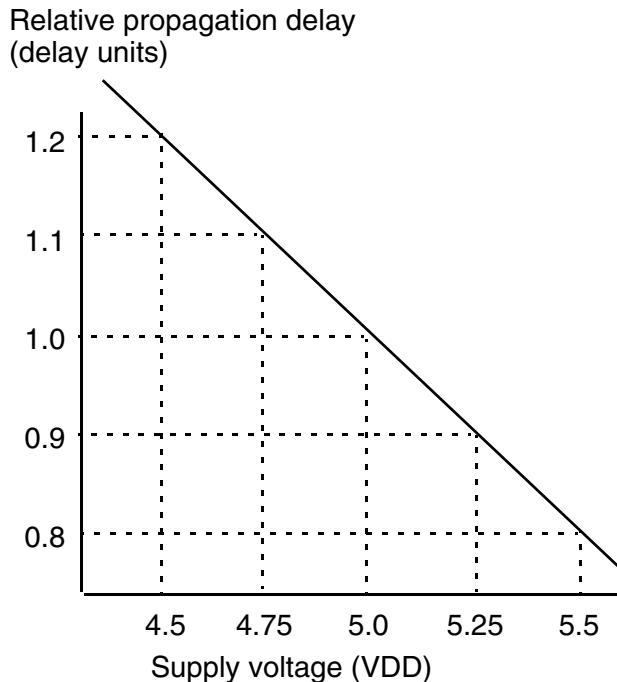
[Figure 6-2](#) illustrates the relationship between supply voltage and propagated delay for a typical technology library.

[Figure 6-2](#) shows that as voltage increases, the relative propagation delay scaling decreases. For example, at 5.25 volts, the delay scaling is 0.90. Determine the k-factors related to voltage by calculating the slope of the line as follows:

$$\begin{aligned}k_{\text{volt}} &= \text{delay scaling units} / \text{volt} \\&= (0.80 - 1.20) / (5.50 - 4.50) \\&= -0.4\end{aligned}$$

If the slope of the line is -0.4 delay units per volt, the value of the k_{volt} delay factor is -0.4 .

Figure 6-2 Propagation Delay as a Function of Supply Voltage



Calculating Temperature Delay Factors

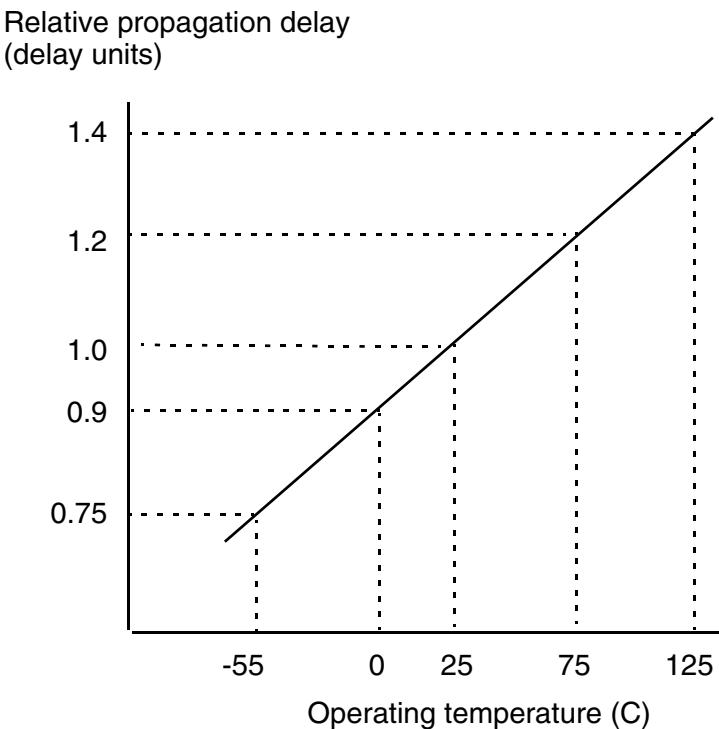
[Figure 6-3](#) illustrates the relationship between operating temperature and propagated delay for a typical technology library. The figure shows that the scaling factor for the delays in the timing equation increases as temperature increases.

Determine the k-factor values related to temperature by calculating the slope of the line as follows:

$$\begin{aligned}k_{\text{temp}} &= \text{delay scaling units / degree Centigrade} \\&= (1.4 - 0.9) / (125 - 0) \\&= 0.004\end{aligned}$$

If the slope of the line is 0.004 delay units per degree centigrade, the value of the `k_temp` delay factor is 0.004.

Figure 6-3 Propagation Delay as a Function of Temperature



Assigning Process Delay Factors

You scale for process by arbitrarily assigning numbers based on the effects of fabrication on timing. There are usually three process-scaling factors available for a library: best case (value less than 1), nominal (value equal or close to 1.0), and worst case (value greater than 1).

Because process-scaling factors are unitless, no graphical representation or slope value is used for process k-factors. The portions of the delay equation affected by process scaling should have a value of 1.

Setting Combined Scaling Factor

According to the technology, you might want to calculate a single scaling factor value to account for the voltage, temperature, and process effects. To implement this type of scaling,

1. Set k-factors for voltage and temperature to 0, which nullifies the scaling effects of voltage and temperature.
2. Set the k-factors for the parts of the delay equation you want to scale to 1.0.
3. Set the `nom_process` attribute value to 1.0. In this case, the `nom_voltage` and `nom_temperature` attribute settings do not matter, because both the voltage and temperature scaling effects are ignored.
4. In the `operating_conditions` groups, set the combined process, temperature, and voltage scaling factor.

Pin and Wire Capacitance Factors

The pin and wire capacitance factors scale the capacitance of a pin or wire according to process, temperature, and voltage variations. Each attribute gives the multiplier for a certain portion of the capacitance of a pin or a wire. In the following syntax, *multiplier* is a floating-point number:

`k_process_pin_cap : multiplier ;`

Scaling factor applied to pin capacitance to model process variation.

`k_process_wire_cap : multiplier ;`

Scaling factor applied to wire capacitance to model process variation.

`k_temp_pin_cap : multiplier ;`

Scaling factor applied to pin capacitance to model temperature variation.

`k_temp_wire_cap : multiplier ;`

Scaling factor applied to wire capacitance to model temperature variation.

`k_volt_pin_cap : multiplier ;`

Scaling factor applied to pin capacitance to model voltage variation.

`k_volt_wire_cap : multiplier ;`

Scaling factor applied to wire capacitance to model voltage variation.

Scaling Factors Associated With the Nonlinear Delay Model

The CMOS nonlinear delay model scaling factors scale the delay based on the variation in process, temperature, and voltage. The following scaling factors apply to the CMOS nonlinear delay model:

- k_process_cell_rise
- k_temp_cell_rise
- k_volt_cell_rise
- k_process_cell_fall
- k_temp_cell_fall
- k_volt_cell_fall
- k_process_rise_propagation
- k_temp_rise_propagation
- k_volt_rise_propagation
- k_process_fall_propagation
- k_temp_fall_propagation
- k_volt_fall_propagation
- k_process_rise_transition
- k_temp_rise_transition
- k_volt_rise_transition
- k_process_fall_transition
- k_temp_fall_transition
- k_volt_fall_transition

Specifying Nonlinear Delay Tables

The two methods for specifying nonlinear delay tables are:

Method 1

Use rise or fall propagation with rise or fall transition tables in which the total cell delay is the sum of propagation delay and transition delay. Use these attributes:

`k_process_rise_propagation`, `k_temp_rise_propagation`,
`k_volt_rise_propagation`, and so on.

Method 2

Use the cell-rise and cell-fall delay tables in which the transition delay is the index into the cell-delay table. Use these attributes: `k_process_cell_rise`, `k_temp_cell_rise`, `k_volt_cell_rise`, and so on.

You cannot use these k-factors interchangeably. For example, `cell_rise` delay is not scaled by `k_process_rise_propagation`, `k_temp_rise_propagation`, and so on; the `rise_propagation` delay is not scaled by `k_process_cell_rise`, and so on.

7

Defining Core Cells

Cell descriptions are a major part of a technology library. They provide information about the area, function, and timing of each component in an ASIC technology.

Defining core cells for CMOS technology libraries involves the following concepts and tasks described in this chapter:

- [Defining cell Groups](#)
- [Defining pin Groups](#)
- [Defining Bused Pins](#)
- [Defining Signal Bundles](#)
- [Defining Layout-Related Multibit Attributes](#)
- [Defining Multiplexers](#)
- [Defining Decoupling Capacitor Cells, Filler Cells, and Tap Cells](#)

Defining cell Groups

A `cell` group defines a single cell in the technology library.

A `model` group describes limited hierarchical interconnections.

This section discusses the attributes in a `cell` group and a `model` group.

For information about groups within a `cell` or `model` group, see the following sections in this chapter:

- “[Defining pin Groups](#)” on page 7-17
- “[Defining Bused Pins](#)” on page 7-58
- “[Defining Signal Bundles](#)” on page 7-65

See [Chapter 9, “Defining Test Cells,”](#) for a test cell with a `test_cell` group. See [Example 7-1 on page 7-13](#) for an example cell description.

cell Group

The `cell` group statement gives the name of the cell being described. It appears at the `library` group level, as shown here:

```
library (lib_name) {  
    ...  
    cell( name ) {  
        ... cell description ...  
    }  
    ...  
}
```

Use a *name* that corresponds to the name the ASIC vendor uses for the cell. When naming cells, remember that names are case-sensitive. For example, the cell names, AND2, and2, and And2 are all different. Cell names beginning with a number must be enclosed in quotation marks. These rules apply to all user-defined names in the Library Compiler tool.

To create the `cell` group for the AND2 cell, use this syntax:

```
cell( AND2 ) {  
    ... cell description ...  
}
```

To describe a CMOS `cell` group, you use the `type` group and these attributes:

- `area`
- `bundle` ([See “Defining Signal Bundles” on page 7-65.](#))

- bus (See “[Defining Bused Pins](#)” on page 7-58.)
- cell_footprint
- clock_gating_integrated_cell
- contention_condition
- dont_fault
- dont_touch
- dont_use
- is_clock_gating_cell
- map_only
- pad_cell
- pad_type
- pin_equal
- pin_opposite
- preferred
- use_for_size_only
- short

area Attribute

This attribute specifies the cell area.

Example

```
area : 2.0;
```

For unknown or undefined (black box) cells, the `area` attribute is optional. Unless a cell is a pad cell, it should have an `area` attribute. Pad cells should be given an area of 0.0, because they are not used as internal gates.

The Library Compiler tool issues a warning for each cell that does not have an `area` attribute.

cell_footprint Attribute

This attribute assigns a footprint class to a cell.

Example

```
cell_footprint : 5MIL ;
```

Characters in the string are case-sensitive.

Use this attribute to assign the same footprint class to all cells that have the same layout boundary. Cells with the same footprint class are considered interchangeable and can be swapped during in-place optimization.

clock_gating_integrated_cell Attribute

An integrated clock-gating cell is a cell that you or your library developer creates to use especially for clock gating. The cell integrates the various combinational and sequential elements of a clock gate into a single cell that is compiled into gates and located in the technology library.

Consider using an integrated clock-gating cell if you are experiencing timing problems caused by the introduction of randomly chosen logic on your clock line.

Use the `clock_gating_integrated_cell` attribute to specify a value that determines the integrated cell functionality to be used by the clock-gating tools.

Syntax

```
clock_gating_integrated_cell : generic|value_id;
```

`generic`

When you specify the value `generic`, the actual type of clock gating integrated cell structure is determined by accessing the function specified on the library pin.

The Library Compiler tool can determine the actual clock gating structure by accessing the latch groups along with the function attribute specified on the cell.

Note:

Statetables and state functions should not be used. Use latch groups with function groups instead.

`value`

A concatenation of up to four strings that describe the cell's functionality to the clock-gating tools:

- The first string specifies the type of sequential element you want. The options are latch-gating logic and none.
- The second string specifies whether the logic is appropriate for rising- or falling-edge-triggered registers. The options are `posedge` and `negedge`.
- The third (optional) string specifies whether you want test-control logic located before or after the latch or not at all. The options for cells set to latch are `precontrol` (before), `postcontrol` (after), or `no entry`. The options for cells set to no gating logic are `control` and `no entry`.
- The fourth (optional) string, which exists only if the third string does, specifies whether you want observability logic or not. The options are `obs` and `no entry`.

Example

```
clock_gating_integrated_cell : "latch_posedge_precontrol_obs" ;
```

[Table 7-1](#) lists some example values for the `clock_gating_integrated_cell` attribute:

Table 7-1 Values for the `clock_gating_integrated_cell` Attribute

When the value is	The integrated cell must contain
<code>latch_negedge</code>	Latch-based gating logic. Logic appropriate for falling-edge-triggered registers.
<code>latch_posedge_postcontrol</code>	Latch-based gating logic. Logic appropriate for rising-edge-triggered registers. Test-control logic located after the latch.
<code>latch_negedge_precontrol</code>	Latch-based gating logic. Logic appropriate for falling-edge-triggered registers. Test-control logic located before the latch.
<code>none_posedge_control_obs</code>	Latch-free gating logic. Logic appropriate for rising-edge-triggered registers. Test-control logic (no latch). Observability logic.

Setting Pin Attributes for an Integrated Cell

The clock-gating tool requires that you set the pins of your integrated cells by using the attributes listed in [Table 7-2](#). Setting some of the pin attributes, such as those for test and observability, is optional.

Table 7-2 Pin Attributes for Integrated Clock-Gating Cells

Integrated cell pin name	Data direction	Required Library Compiler attribute
clock	in	clock_gate_clock_pin
enable	in	clock_gate_enable_pin
test_mode or scan_enable	in	clock_gate_test_pin
observability	out	clock_gate_obs_pin
enable_clock	out	clock_gate_out_pin

For details about these pin attributes, see the following sections:

- “[clock_gate_clock_pin Attribute](#)” on page 7-20
- “[clock_gate_enable_pin Attribute](#)” on page 7-20
- “[clock_gate_obs_pin Attribute](#)” on page 7-21
- “[clock_gate_out_pin Attribute](#)” on page 7-21
- “[clock_gate_test_pin Attribute](#)” on page 7-21

For more details about the `clock_gating_integrated_cell` attribute and the corresponding pin attributes, see [Chapter 12, “Modeling Power and Electromigration,”](#) and the *Power Compiler User Guide*.

Setting Timing for an Integrated Cell

You set both the setup and hold arcs on the enable pin by setting the `clock_gate_enable_pin` attribute for the integrated cell to `true`. The setup and hold arcs for the cell are determined by the edge values you enter for the `clock_gating_integrated_cell` attribute. [Table 7-3](#) lists the edge values and the corresponding setup and hold arcs.

Table 7-3 Values of the `clock_gating_integrated_cell` Attributes

Value	Setup arc	Hold arc
<code>latch_posedge</code>	<code>rising</code>	<code>rising</code>
<code>latch_negedge</code>	<code>falling</code>	<code>falling</code>
<code>none_posedge</code>	<code>falling</code>	<code>rising</code>
<code>none_negedge</code>	<code>rising</code>	<code>falling</code>

For details about setting timing for an integrated cell, see [Chapter 12, “Modeling Power and Electromigration”](#).

contention_condition Attribute

The DFT Compiler tool uses the `contention_condition` attribute to specify the contention conditions for a cell.

Contention is a clash of 0 and 1 signals. In certain cells, it can be a forbidden condition and cause circuits to short.

Example

```
contention_condition : "!ap * an" ;
```

dont_fault Attribute

This attribute is used by DFT Compiler. It is a string attribute that you can set on a library cell or pin.

Example

```
dont_fault : sa0;
```

If you set this attribute on a library cell to `sa0`, instances of the cell in the design are not modeled with stuck-at-0 faults during fault modeling. Similarly, if you set this attribute on a library cell to `sa1`, instances of the cell in the design are not modeled with stuck-at-1 faults.

Setting this attribute on a cell to `sa01` means that the cell is not modeled with either stuck-at-0 or stuck-at-1 faults. The same values are allowed on pins and mean that the pin is not modeled with the specified stuck-at fault.

dont_touch Attribute

If you do not want a cell removed during optimization, use the `dont_touch` attribute. When you set this attribute to `true`, all instances of the cell must remain in the network.

Example

```
dont_touch : true;
```

When it encounters the cell during optimization, the Design Compiler tool works around the cell and does not replace it. You can apply the `dont_touch` attribute to a special clock-distribution cell.

In addition to defining `dont_touch` in your technology library, you can also apply the `dont_touch` attribute to a cell with the `set_dont_touch` command from `dc_shell`. You cannot remove a cell with the `dont_touch` attribute from `dc_shell`.

dont_use Attribute

If you do not want a cell added to a design during optimization, set this attribute to `true`. For example, you could specify `dont_use` for an I/O cell that you do not want the Design Compiler tool to put into the core of your design.

Example

```
dont_use : true;
```

In addition to defining `dont_use` in your technology library, you can also apply the `dont_use` attribute to a cell with the `set_dont_use` command from `dc_shell`.

is_clock_gating_cell Attribute

The `is_clock_gating_cell` attribute identifies cells that are used for clock gating. the Design Compiler tool uses this attribute when it compiles a design containing gated clocks that are introduced by the Power Compiler tool.

Set this attribute only on 2-input AND, NAND, OR, and NOR gates; inverters; buffers; and 2-input D latches. Valid values for this attribute are `true` and `false`.

Example

```
is_clock_gating_cell : true;
```

To use a cell exclusively during the mapping of clock-gating circuits, set the `is_clock_gating_cell` attribute to `true` and the `dont_touch` attribute to `true`.

See “[Defining pin Groups](#)” on page [7-17](#) for information about designating clock and enable ports on clock gates. For additional information about clock gating, see the *Power Compiler User Guide*.

is_macro_cell Attribute

The `is_macro_cell` attribute identifies whether a cell is a macro cell. If the attribute is set to `true`, the cell is a macro cell. If it is set to `false`, the cell is not a macro cell.

Example

```
is_macro_cell : true;
```

is_memory_cell Attribute

The `is_memory_cell` attribute identifies whether a cell is a memory cell. If the attribute is set to `true`, the cell is a memory cell, if it is set to `false`, the cell is not a memory cell.

Example

```
is_memory_cell : true;
```

map_only Attribute

If this attribute has the value `true`, the cell is excluded from logic-level optimization during compilation.

Example

```
map_only : true;
```

In addition to defining `map_only` in a `cell` group, you can also apply the `map_only` attribute to a cell by using the `set_map_only` command from `dc_shell`.

pad_cell Attribute

The `pad_cell` attribute in a `cell` group identifies the cell as a pad. The Design Compiler tool filters out pads during core optimization and treats them differently during technology translation.

Example

```
pad_cell : true ;
```

If the `pad_cell` attribute is included in a cell definition, at least one pin in the cell must have an `is_pad` attribute. See “[is_pad Attribute](#)” on page [7-53](#).

If more than one pad cell is used to build a logical pad, put this attribute in the cell definitions of all the component pad cells:

```
auxiliary_pad_cell : true ;
```

If you omit the `pad_cell` or `auxiliary_pad_cell` attribute, the cell is treated as an internal core cell.

Note:

A cell with an `auxiliary_pad_cell` attribute can also be used within the core; a pull-up or pull-down cell is an example of such a cell.

pad_type Attribute

This attribute identifies a pad cell or auxiliary pad cell that requires special treatment. The only type of pad cell supported is `clock`, which identifies a pad cell as a clock driver.

Example

```
pad_type : clock ;
```

pin_equal Attribute

This attribute describes a group of logically equivalent input or output pins in the cell.

Example

```
pin_equal : ("Y Z") ;
```

The Design Compiler tool automatically determines the equivalent pins for cells that have `function` attributes. See “[function Attribute](#)” on page [7-47](#) for more information.

Note:

Use the `pin_equal` attribute only in cells without function information or when you want to define required input values.

pin_opposite Attribute

This attribute describes functionally opposite (logically inverse) groups of pins in a cell. The `pin_opposite` attribute also incorporates the functionality of `pin_equal`.

Example

```
pin_opposite("Q1 Q2 Q3", "QB1 QB2") ;
```

In this example, Q1, Q2, and Q3 are equal; QB1 and QB2 are equal; and the pins of the first group are opposite to the pins of the second group.

The Design Compiler tool automatically determines the opposite pins for cells that have `function` attributes. See “[function Attribute](#)” on page [7-47](#) for more information.

Note:

Use the `pin_opposite` attribute only in cells without function information or when you want to define required inputs.

preferred Attribute

Setting this attribute to value `true` indicates that the cell is the preferred replacement during the gate-mapping phase of optimization.

Example

```
preferred : true ;
```

The Design Compiler tool chooses cells with the `preferred` attribute over other cells with the same function if the standard cell offers no optimization advantage.

You can apply the `preferred` attribute to a cell with preferred timing or area attributes. For example, in a set of 2-input NAND gates, you might want to use gates with higher drive strengths wherever possible. This practice is useful primarily in design translation.

use_for_size_only Attribute

You use this attribute to specify the criteria for sizing optimization. You set this attribute on a cell at the cell level. When this attribute is set on a cell, the Design Compiler tool does not map to that cell. As a result, a designer has to instantiate those cells.

Example

```
library(lib1){  
    cell(cell1){  
        area : 14 ;  
        use_for_size_only : true ;  
        pin(A){  
            ...  
        }  
        ...  
    }  
}
```

short Attribute

The `short` complex attribute lists the shorted ports or pins that are connected by metal or polytrace. You must specify at least two pin names in the list.

The most common example of a shorted port is a feedthrough, where an input port is directly connected to an output port. Another example is two output ports that fan out from the same gate.

To model multiple sets of feedthrough pins in a cell, specify multiple `short` attributes. For example, if pins a and b are shorted to create a feedthrough, and pins c and d are shorted to create another feedthrough, use:

```
short (a,b);  
short (c,d);
```

However, if pin a is shorted with pin b, and pin a is also shorted with pin c, that is, pins a, b, and c are shorted, use:

```
short (a,b,c);
```

This is equivalent to the following syntax:

```
short (a,b);  
short (a,c);
```

Note:

The tool recognizes a cell where the `short` attribute is defined as a feethrough cell, and automatically marks the cell with the `is_feed_through_cell` attribute set to `true`.

The tool also recognizes a pin specified in the `short` attribute as a feedthrough pin, and automatically marks the pin with the `is_feed_through_pin` attribute set to `true`.

Do not specify the `related_power_pin` or the `related_ground_pin` attribute for a feedthrough pin, that is, a pin listed in the `short` attribute. The Power Compiler tool can connect the feedthrough pin to a desired power domain.

Syntax

```
short("pinname_list") ;
```

Example

```
library(short) {  
    cell(feedthru) {  
        ...  
        /* y and a are feed through pins */  
        short(y, a);  
        ...  
        pin(y) {  
            direction : output;  
            timing() {
```

```

        related_pin : a;
        ...
    }
}
pin(a) {
    direction : input;
    ...
}
...
}
}

```

Note:

You can also model feedthrough cells by defining the `short` attribute in a `model` group.

type Group

The `type` group, when defined within a cell, is a type definition local to the cell. It cannot be used outside of the cell.

Example

```

type (bus4) {
    base_type : array;
    data_type : bit;
    bit_width : 4;
    bit_from : 0;
    bit_to : 3;
}

```

cell Group Example

[Example 7-1](#) shows cell definitions that include some of the CMOS cell attributes described in this section.

Example 7-1 cell Group Example

```

library (cell_example){
    date : "August 14, 2015";
    revision : 2015.03;
    cell (inout){
        pad_cell : true;
        dont_use : true;
        dont_fault : sa0;
        dont_touch : true;
        area : 0; /* pads do not normally consume internal
                    core area */
        cell_footprint : 5MIL;
        pin (A) {

```

```
    direction : input;
    capacitance : 0;
}
pin (Z) {
    direction : output;
    function : "A";
    timing () {
        ...
    }
}
cell(inverter_med){
    area : 3;
    preferred : true;
    pin (A) {
        direction : input;
        capacitance : 1.0;
    }
    pin (Z) {
        direction : output;
        function : "A' ";
        timing () {
            ...
        }
    }
}
cell(nand){
    area : 4;
    pin(A) {
        direction : input;
        capacitance : 1;
        fanout_load : 1.0;
    }
    pin(B) {
        direction : input;
        capacitance : 1;
        fanout_load : 1.0;
    }

    pin (Y) {
        direction : output;
        function : "(A * B)' ";
        timing() {
            ...
        }
    }
}
cell(buff1){
    area : 3;
    pin (A) {
        direction : input;
        capacitance : 1.0;
    }
}
```

```

    pin (Y) {
        direction : output;
        function : "A ";
        timing () {
            ...
        }
    }
} /* End of Library */

```

mode_definition Group

A mode_definition group declares a mode group that contains several timing mode values.

PrimeTime can enable each timing arc, based on the current mode of the design, which results in different timing for different modes. You can optionally put a condition on a mode value. When the condition is true, the mode group takes that value.

Syntax

```

cell(name_string) {
    mode_definition(name_string) {
        mode_value(name1) {
            when : "Boolean expression" ;
            sdf_cond : "sdf_expression_string" ;
        }
        mode_value(name_string) {
            when : "Boolean expression" ;
            sdf_cond :"Boolean expression" ;
        }
    }
}

```

Group Statement

```
mode_value (name_string) { }
```

Specifies the condition that a timing arc depends on to activate a path.

mode_value Group

The mode_value group contains several mode values within a mode group. You can optionally put a condition on a mode value. When the condition is true, the mode group takes that value.

Syntax

```
mode_value (name_string) { }
```

Simple Attributes

```
when : "Boolean expression" ;
sdf : "Boolean expression" ;
```

when Simple Attribute

The `when` attribute specifies the condition that a timing arc depends on to activate a path. The valid value is a Boolean expression.

Syntax

```
when : "Boolean expression" ;
```

Example

```
when: !R;
```

sdf_cond Simple Attribute

The `sdf_cond` attribute supports Standard Delay Format (SDF) file generation and condition matching during back-annotation; however, PrimeTime does not use this attribute for back-annotation.

Syntax

```
sdf_cond : "Boolean expression" ;
```

Example

```
sdf_cond: "R == 0";
```

[Example 7-2](#) shows a `mode_definition` description.

Example 7-2 mode_definition Description

```
cell(example_cell) {
...
    mode_definition(rw) {
        mode_value(read) {
            when : "R";
            sdf_cond : "R == 1";
        }
        mode_value(write) {
            when : "!R";
            sdf_cond : "R == 0";
        }
    }
}
```

Defining pin Groups

For each pin in a cell, the `cell` group must contain a description of the pin characteristics. You define pin characteristics in a `pin` group within the `cell` group.

A `pin` group often contains a `timing` group and an `internal_power` group.

For more information about `timing` groups, see [Chapter 11, “Timing Arcs”](#).

For more information about the `internal_power` group, see [Chapter 12, “Modeling Power and Electromigration”](#).

pin Group

You can define a `pin` group within a `cell`, `test_cell`, model, or `bus` group.

You can also define a `pin` group within a `bundle` group, however, the Design Compiler tool does not accept a `pin` group in the `bundle` group.

```
library (lib_name) {  
  ...  
  cell (cell_name) {  
    ...  
    pin ( name | name_list ) {  
      ... pin group description ...  
    }  
  }  
  cell (cell_name) {  
    ...  
    bus (bus_name) {  
      ... bus group description ...  
    }  
    bundle (bundle_name) {  
      ... bundle group description ...  
    }  
    pin ( name | name_list ) {  
      ... pin group description ...  
    }  
  }  
}
```

See [“Defining Bused Pins” on page 7-58](#) for descriptions of `bus` groups. See [“Defining Signal Bundles” on page 7-65](#) for descriptions of `bundle` groups.

The `pin` groups are also valid within `test_cell` groups. They have different requirements from `pin` groups in `cell`, `bus`, or `bundle` groups. See [“Pins in the test_cell Group” on page 9-3](#) for specific information and restrictions on describing test pins.

All pin names within a single `cell`, `bus`, or `bundle` group must be unique. Names are case-sensitive: pins named `A` and `a` are different pins.

Pin names beginning with a number must be enclosed in quotation marks; for the exceptions to this rule, see the “[function Attribute](#)” on page [7-47](#) and the information about the `related_pin` attribute in [Chapter 11, “Timing Arcs”](#).

Also, pin names that include a special character, such as a greater-than symbol (>), or a lesser-than symbol (<), must be enclosed in quotation marks.

Note:

Although netlists can be written out in either implicit or explicit formats, the Library Compiler tool supports only the implicit format.

Implicit, or order-based, netlists attach nets to components according to the order in which they are defined in the netlist. The order in which the pins are listed in the descriptions of the cells is the order in which they are referenced when writing (and possibly reading) a netlist. Therefore, the Library Compiler cell descriptions must have output pins listed before input pins.

Explicit netlists include the pin name and the net connected to the pin, and the order is not important. The Library Compiler tool does not support this format.

You can describe pins with common attributes in a single `pin` group. If a cell contains two pins with different attributes, two separate `pin` groups are required. Grouping pins with common technology attributes can significantly reduce the size of a cell description that includes many pins.

Do not confuse defining multiple pins in a single `pin` group with defining a bus or bundle. See “[Defining Bused Pins](#)” on page [7-58](#) and “[Defining Signal Bundles](#)” on page [7-65](#).

In the following example, the AND cell has two pins: A and B.

```
cell (AND) {
    area : 3 ;
    pin (A) {
        direction : input ;
        capacitance : 1 ;
    }
    pin (B) {
        direction : input ;
        capacitance : 1 ;
    }
}
```

Because pins A and B have the same attributes, the cell can also be described as

```
cell (AND) {
    area : 3 ;
    pin (A,B) {
        direction : input ;
        capacitance : 1 ;
    }
}
```

General pin Group Attributes

To define a pin, use these general `pin` group attributes:

- `capacitance`
- `clock_gate_clock_pin`
- `clock_gate_enable_pin`
- `clock_gate_obs_pin`
- `clock_gate_out_pin`
- `clock_gate_test_pin`
- `complementary_pin`
- `connection_class`
- `direction`
- `dont_fault`
- `driver_type`
- `fall_capacitance`
- `fault_model`
- `inverted_output`
- `is_analog`
- `pin_func_type`
- `rise_capacitance`
- `steady_state_resistance`
- `test_output_only`

For a complete list and descriptions for all the attributes and groups that you can specify in a pin group, see Chapter 3, “pin Group Description and Syntax,” in the *Synopsys Logic Library Reference Manual*.

capacitance Attribute

The `capacitance` attribute defines the load of an input, output, inout, or internal pin. The load is defined with a floating-point number, in units consistent with other capacitance specifications throughout the library. Typical units of measure for capacitance include picofarads and standardized loads.

Example

The following example defines the A and B pins in an AND cell, each with a capacitance of one unit.

```
cell (AND) {
    area : 3 ;
    pin (A,B) {
        direction : input ;
        capacitance : 1 ;
    }
}
```

If the `timing` groups in a cell include the output-pin capacitance effect in the intrinsic-delay specification, do not specify capacitance values for the cell's output pins.

clock_gate_clock_pin Attribute

The `clock_gate_clock_pin` attribute identifies an input pin connected to a clock signal. The Design Compiler tool uses the `clock_gate_clock_pin` attribute when it compiles a design containing gated clocks that were introduced by the Power Compiler tool.

Valid values for this attribute are `true` and `false`.

Example

```
clock_gate_clock_pin : true;
```

See “[clock_gating_integrated_cell Attribute](#)” on page 7-4 for more information about identifying pins on integrated clock-gating cells. For additional information about integrated clock-gating, see the *Power Compiler User Guide*.

clock_gate_enable_pin Attribute

The Design Compiler tool uses the `clock_gate_enable_pin` attribute when it compiles a design containing gated clocks that were introduced by the Power Compiler tool.

The `clock_gate_enable_pin` attribute identifies an input port connected to an enable signal for nonintegrated clock-gating cells and integrated clock-gating cells.

Valid values for this attribute are `true` and `false`. A `true` value labels the input port pin connected to an enable signal for nonintegrated and integrated clock-gating cells. A `false` value labels the input port pin connected to an enable signal as *not* for nonintegrated and integrated clock-gating cells.

Example

```
clock_gate_enable_pin : true;
```

For nonintegrated clock-gating cells, you can set the `clock_gate_enable_pin` attribute to true on only one input port of a 2-input AND, NAND, OR, or NOR gate. If you do so, the other input port is the clock.

See “[clock_gating_integrated_cell Attribute](#)” on page 7-4 for more information about identifying pins on integrated clock-gating cells. For additional information about integrated clock gating, see the *Power Compiler User Guide*.

clock_gate_obs_pin Attribute

The `clock_gate_obs_pin` attribute identifies an output port connected to an observability signal. The Design Compiler tool uses the `clock_gate_obs_pin` attribute when it compiles a design containing gated clocks that were introduced by the Power Compiler tool.

Valid values for this attribute are `true` and `false`. A `true` value labels the pin as an observability pin. A `false` value labels the pin as not an observability pin.

Example

```
clock_gate_obs_pin : true;
```

See “[clock_gating_integrated_cell Attribute](#)” on page 7-4 for more information about identifying pins on integrated clock-gating cells. For additional information about integrated clock gating, see the *Power Compiler User Guide*.

clock_gate_out_pin Attribute

The `clock_gate_out_pin` attribute identifies an output port connected to an `enable_clock` signal. The Design Compiler tool uses the `clock_gate_out_pin` attribute when it compiles a design containing gated clocks that were introduced by the Power Compiler tool.

Valid values for this attribute are `true` and `false`. A `true` value labels the pin as an out (`enable_clock`) pin. A `false` value labels the pin as *not* an out pin.

Example

```
clock_gate_out_pin : true;
```

See “[clock_gating_integrated_cell Attribute](#)” on page 7-4 for more information about identifying pins on integrated clock-gating cells. For additional information about integrated clock gating, see the *Power Compiler User Guide*.

clock_gate_test_pin Attribute

The `clock_gate_test_pin` attribute identifies an input port connected to a `test_mode` or `scan_enable` signal. The Design Compiler tool uses the `clock_gate_test_pin` attribute when it compiles a design containing gated clocks that were introduced by the Power Compiler tool.

Valid values for this attribute are `true` and `false`. A `true` value labels the pin as a test (`test_mode` or `scan_enable`) pin. A `false` value labels the pin as not a test pin.

Example

```
clock_gate_test_pin : true;
```

See “[clock_gating_integrated_cell Attribute](#)” on page 7-4 for more information about identifying pins on integrated clock-gating cells. For additional information about integrated clock gating, see the *Power Compiler User Guide*.

complementary_pin Simple Attribute

The `complementary_pin` attribute, which works only with DFT Compiler, supports differential I/O.

Differential I/O assumes the following:

- When the noninverting pin equals 1 and the inverting pin equals 0, the signal gets logic 1.
- When the noninverting pin equals 0 and the inverting pin equals 1, the signal gets logic 0.

The entry for this attribute identifies the differential input inverting pin with which the noninverting pin is associated and from which it inherits timing information and associated attributes.

The Library Compiler tool automatically generates a connection class differential without your having to set it with the `connection_class` attribute. For information about the `connection_class` attribute, see “[connection_class Simple Attribute](#).”

Syntax

```
complementary_pin : "string" ;
```

string

Identifies the differential input data inverting pin whose timing information and associated attributes the noninverting pin inherits. Only one input pin is modeled at the cell level. The associated differential inverting pin is defined in the same `pin` group.

For details on the `fault_model` attribute used to define the value when both the complementary pin and the pin that it complements are driven to the same value, see “[fault_model Simple Attribute](#)” on page 7-31.

Example

```
cell (diff_buffer) {
    ...
    pin (A) { /* noninverting pin /
        direction : input ;
        complementary_pin : "DiffA" /* inverting pin /
```

```

    }
}

```

connection_class Simple Attribute

The `connection_class` attribute lets you specify design rules for connections between cells.

Example

```
connection_class : "internal";
```

Only pins with the same connection class can be legally connected. For example, you can specify that clock input must be driven by clock buffer cells or that output pads can be driven only by high-drive pad driver cells between the internal logic and the pad. To do this, you assign the same connection class to the pins that must be connected. For the pad example, you attach a given connection class to the pad driver output and the pad input. This attachment makes it invalid to connect another type of cell to the pad.

The Design Compiler tool recognizes two special connection classes: universal and default.

- The universal connection class matches all other connection classes, so a pin with this class can be connected to another pin.
- The Design Compiler tool assigns the default connection class to all pins that do not have a connection class assigned to them. A default pin can be connected only to another default pin.

You can define a default connection class with the `default_connection_class` attribute in the `library` group, as described in the “Library-Level Default Attributes” section in [Chapter 6, “Building Environments”](#). If you do not define a default connection class for your library, the Design Compiler tool assigns the value to the `default_connection_class` attribute.

Nets with multiple drivers, such as buses or wired logic, follow the same connection rules as single-driver nets. All pins must have the same connection class.

[Example 7-3](#) uses connection classes. The `output_pad` cell can be driven only by the `pad_driver` cell. The pad driver’s input can be connected to internal core logic, because it has the internal connection class.

[Example 7-4](#) shows the use of multiple connection classes for a single pin. The `high_drive_buffer` cell can drive internal core cells and pad cells, whereas the `low_drive_buffer` cell can drive only internal cells.

Example 7-3 Connection Class Example

```
default_connection_class : "default" ;
cell (output_pad) {
    pin (IN) {
        connection_class : "external_output" ;
    ...
}
```

```
        }
    }
cell (pad_driver) {
    pin (OUT) {
        connection_class : "external_output" ;
    ...
}
    pin (IN) {
        connection_class : "internal" ;
    ...
}
}
```

Example 7-4 Multiple Connection Classes for a Pin

```
cell (high_drive_buffer) {
    pin (OUT) {
        connection_class : "internal pad" ;
    ...
}
cell (low_drive_buffer) {
    pin (OUT) {
        connection_class : "internal" ;
    ...
}
cell (pad_cell) {
    pin (IN) {
        connection_class : "pad" ;
    ...
}
cell (internal_cell) {
    pin (IN) {
        connection_class : "internal" ;
    ...
}
}
```

direction Attribute

Use this attribute to specify to the Design Compiler tool whether the pin being described is an input, output, internal, or bidirectional pin.

Example

```
direction : output;
```

For a description of the effect of pin direction on path tracing and timing analysis, see the *Design Compiler Optimization Reference Manual*.

dont_fault Attribute

DFT Compiler uses the `dont_fault` attribute. It is a string attribute that you can set on a library cell or pin.

Example

```
dont_fault : sa0;
```

If you set this attribute on a library cell to `sa0`, instances of the cell in the design are not modeled with stuck-at-0 faults during fault modeling. Similarly, if you set this attribute on a library cell to `sa1`, instances of the cell in the design are not modeled with stuck-at-1 faults. Setting this attribute on a cell to `sa01` means that the cell is not modeled with either stuck-at-0 or stuck-at-1 faults. The same values are allowed on pins and mean that the pins are not modeled with the specified stuck-at fault.

driver_type Attribute

Use the optional `driver_type` attribute to modify the signal on a pin. This attribute specifies a signal mapping mechanism that supports the signal transitions performed by the circuit.

The `driver_type` attribute tells the application tool to use a special pin-driving configuration for the pin during simulation. A pin without this attribute has normal driving capability by default.

A driver type can be one or more of the following:

pull_up

The pin is connected to DC power through a resistor. If it is a three-state output pin and it is in the Z state, its function is evaluated as a resistive 1 (H). If it is an input or inout pin and the node to which it is connected is in the Z state, it is considered an input pin at logic 1 (H). For a pull-up cell, the pin stays constantly at logic 1 (H).

pull_down

The pin is connected to DC ground through a resistor. If it is a three-state output pin and it is in the Z state, its function is evaluated as a resistive 0 (L). If it is an input or inout pin and the node to which it is connected is in the Z state, it is considered an input pin at logic 0 (L). For a pull-down cell, the pin stays constantly at logic 0 (L).

Note:

To alert you that the output driving the input could be affected by the pull-up or pull-down driver, the Library Compiler tool issues a warning whenever an input pin has a pull-up or pull-down driver.

bus_hold

The pin is a bidirectional pin on a bus holder cell. The pin holds the last logic value present at that pin when no other active drivers are on the associated net. Pins with this driver type cannot have function or three_state statements.

open_drain

The pin is an output pin without a pull-up transistor. Use this driver type only for off-chip output or inout pins representing pads. The pin goes to high impedance (Z) when its function is evaluated as logic 1.

open_source

The pin is an output pin without a pull-down transistor. Use this driver type only for off-chip output or inout pins representing pads. The pin goes to high impedance (Z) when its function is evaluated as logic 0.

resistive

The pin is an output pin connected to a controlled pull-up or pull-down driver with a control port (input). When the control port is disabled, the pull-up or pull-down driver is turned off and has no effect on the pin. When the control port is enabled, a functional value of 0 evaluated at the pin is turned into a weak 0 (L), a functional value of 1 is turned into a weak 1 (H), but a functional value of Z is not affected.

resistive_0

The pin is an output pin connected to DC power through a pull-up driver that has a control port (input). When the control port is disabled, the pull-up driver is turned off and has no effect on the pin. When the control port is enabled, a functional value of 1 evaluated at the pin is turned into a weak 1 (H) but the functional values of 0 and Z are not affected.

resistive_1

The pin is an output pin connected to DC ground through a pull-down driver that has a control port (input). When the control port is disabled, the pull-down driver is turned off and has no effect on the pin. When the control port is enabled, a functional value of 0 evaluated at the pin is turned into a weak 0 (L) but the functional values of 1 and Z are not affected.

Except for inout pins, each pin can have only one `driver_type` attribute. The Library Compiler tool ignores multiple statements and uses the last statement.

Inout pins can have two driver types, one for input and one for output. The only valid combinations are `pull_up` or `pull_down` for input and `open_drain` for output. If you specify only one driver type and it is `bus_hold`, it is used for both input and output. If the single driver type is not `bus_hold`, it is used for output. Specify multiple driver types in one entry in this format:

```
driver_type : "driver_type1 driver_type2" ;
```

Example

This is an example of a pin connected to a controlled pull-up cell that results in a weak 1 when the control port is enabled.

```
function : 1;
driver_type : resistive;
three_state_enable : EN;
```

Interpretation of Driver Types

The driver type specifies one of the following signal modifications:

Resolve the value of Z

These driver types resolve the value of Z on an existing circuit node, implying a constant 0 or 1 signal source. They do not perform a function. Resolution driver types are pull_up, pull_down, and bus_hold.

Transform the signal

These driver types perform an actual function on an input signal, mapping the transition from 0 or 1 to L, H, or Z. Transformation driver types are open_drain, open_source, resistive, resistive_0, and resistive_1.

For output pins, the `driver_type` attribute is applied after the pin's functional evaluation. For input pins, this attribute is applied before the signal is used for functional evaluation. For the signal mapping and pin types for different driver types, see [Table 7-4](#).

Table 7-4 Driver Types

Driver type	Description	Signal mapping	Applicable pin types
pull_up	Resolution	01Z -> 01H	in, out
pull_down	Resolution	01Z -> 01L	in, out
bus_hold	Resolution	01Z -> 01S	inout
open_drain	Transformation	01Z -> 0ZZ	out
open_source	Transformation	01Z -> Z1Z	out
resistive	Transformation	01Z -> LHZ	out
resistive_0	Transformation	01Z -> 0HZ	out
resistive_1	Transformation	01Z -> L1Z	out

Signal Mapping:

0 and 1 represent strong logic 0 and logic 1 values

L represents a weak logic 0 value

H represents a weak logic 1 value

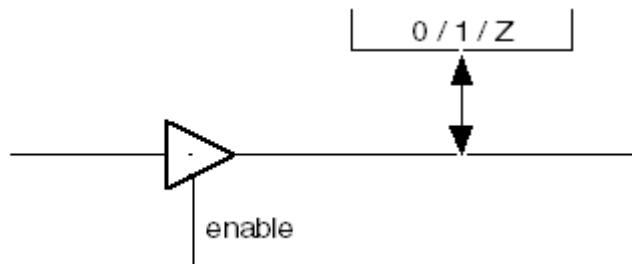
Z represents high impedance

S represents the previous state

Interpreting Bus Holder Driver Type

[Figure 7-1](#) illustrates the 01Z to 01S signal mapping for bus holders.

Figure 7-1 Interpreting bus_hold Driver Type



For bus_holder driver types, a three-state buffer output value of 0 changes the bus value to 0. Similarly, a three-state buffer output value of 1 changes the bus value to 1. However, when the output of the three-state buffer is Z, the bus holds its previous value (S), which can be 0, 1, or Z. In other words, the buffer output value of Z is resolved to the previous value of the bus.

Modeling Pull-Up and Pull-Down Cells

[Figure 7-2](#) shows a pull-up resistor cell.

Figure 7-2 Pull-Up Resistor of a Cell



[Example 7-5](#) is the description of the pull-up resistor cell in [Figure 7-2](#). This cell has a pull-up driver type and no input. Because it is functionless, the Library Compiler tool labels it a black box. This model requires special recognition from the tools that use it. This cell is a pad, but you can omit the pad information for an internal pull-up cell.

Example 7-5 Description of a Pull-Up Cell Transistor

```
cell(pull_up_cell) {
    area : 0;
    auxiliary_pad_cell : true;
    pin(Y) {
        direction : output;
        multicell_pad_pin : true;
        connection_class : "inpad_network";
        driver_type : pull_up;
        pulling_resistance : 10000;
    }
}
```

Example 7-6 describes an output pin with a pull-up resistor and the bidirectional pin on a bus holder cell.

Example 7-6 Pin Driver Type Specifications

```
pin(Y) {
    direction : output ;
    driver_type : pull_up ;
    pulling_resistance : 10000 ;
    function : "IO" ;
    three_state : "OE" ;
}
cell(bus_hold) {
    pin(Y) {
        direction : inout ;
        driver_type : bus_hold ;
    }
}
```

Bidirectional pads can often require one driver type for the output behavior and another associated with the input. For this case, you can define multiple driver types in one `driver_type` attribute:

```
driver_type : "open_drain pull_up" ;
```

Note:

An *n*-channel open-drain pad is flagged with `open_drain`, and a *p*-channel open-drain pad is flagged with `open_source`.

fall_capacitance Simple Attribute

Defines the load for an input and inout pin when its signal is falling.

Setting a value for the `fall_capacitance` attribute requires that a value for the `rise_capacitance` also be set, and setting a value for the `rise_capacitance` requires that a value for the `fall_capacitance` also be set.

Note:

If the Library Compiler tool does not find a `capacitance` attribute for an input or inout pin, it generates a warning message and uses the larger of the value settings for `fall_capacitance` and `rise_capacitance`. If `fall_capacitance` and `rise_capacitance` have not been set for the input pin, the Library Compiler tool uses the value to which the `default_input_pin_cap` attribute is set, and for the inout pin, it uses the value to which the `default_inout_pin_cap` attribute is set.

Syntax

```
fall_capacitance : float ;
```

float

A floating-point number in units consistent with other capacitance specifications throughout the library. Typical units of measure for `fall_capacitance` include picofarads and standardized loads.

Example

The following example defines the A and B pins in an AND cell, each with a `fall_capacitance` of one unit, a `rise_capacitance` of two units, and a capacitance of two units.

```
cell (AND) {
    area : 3 ;
    pin (A,B) {
        direction : input ;
        fall_capacitance : 1 ;
        rise_capacitance : 2 ;
        capacitance : 2 ;
    }
}
```

fault_model Simple Attribute

The differential I/O feature enables an input noninverting pin to inherit the timing information and all associated attributes of an input inverting pin in the same `pin` group designated with the `complementary_pin` attribute.

The optional `fault_model` attribute is used only by the DFT Compiler tool. If you enter a `fault_model` attribute, you must designate the inverted pin associated with the noninverting pin, using the `complementary_pin` attribute.

For details on the `complementary_pin` attribute, see “[complementary_pin Simple Attribute](#)” on page 7-22.

Syntax

`fault_model : "two-value string" ;`

two-value string

Two values that define the value of the differential signals when both inputs are driven to the same value. The first value represents the value when both input pins are at logic 0; the second value represents the value when both input pins are at logic 1. Valid values for the two-value string are any two-value combinations of 0, 1, and x.

If you do not enter a `fault_model` attribute value, the signal pin value goes to x when both input pins are 0 or 1.

Example

```
cell (diff_buffer) {
    ...
    pin (A) { /* noninverting pin /
        direction : input ;
        complementary_pin : ("DiffA")
        fault_model : "1x" ;
    }
}
```

[Table 7-5](#) shows how testing using the DFT Compiler tool interprets the complementary pin values for this example:

Table 7-5 Interpretation of Pin Values

Pin A (noninverting pin)	DiffA (complementary_pin)	Resulting signal pin value
1	0	1
0	1	0
0	0	1
1	1	x

inverted_output Attribute

The `inverted_output` attribute is a Boolean attribute that you can set for any output port. It is a required attribute only for sequential cells.

Set this attribute to false for noninverting output, which is variable1 or IQ for flip-flop or latch groups. Set this attribute to true for inverting output, which is variable2 or IQN for flip-flop or latch groups.

Example

```
pin(Q) {
    function : "IQ";
    internal_node : "IQ";
    inverted_output : false;
}
```

This attribute affects the internal interpretation of the state table format used to describe a sequential cell.

is_analog Attribute

The `is_analog` attribute identifies an analog signal pin as analog so it can be recognized by tools. The valid values for `is_analog` are `true` and `false`. Set the `is_analog` attribute to `true` at the pin level to specify that the signal pin is analog.

Syntax

The syntax for the `is_analog` attribute is as follows:

```
cell (cell_name) {  
    ...  
    pin (pin_name) {  
        is_analog: true | false ;  
        ...  
    }  
}
```

Example

The following example identifies the pin as an analog signal pin.

```
pin(Analog) {  
    direction : input;  
    capacitance : 1.0 ;  
    is_analog : true;  
}
```

pin_func_type Attribute

This attribute describes the functions of a pin.

Example

```
pin_func_type : clock_enable;
```

With the `pin_func_type` attribute, you avoid the checking and modeling caused by incomplete timing information about the enable pin. The information in this attribute defines the clock as the clock-enabling mechanism (that is, the clock-enable pin). This attribute also specifies whether the active level of the enable pin of latches is high or low and whether the active edge of the flip-flop clock is rising or falling.

The `report_lib` command lists the pins of a cell if they have one of these `pin_func_type` values: `active falling`, `active high`, `active low`, `active rising`, or `clock enable`.

rise_capacitance Simple Attribute

This attribute defines the load for an input and inout pin when its signal is rising.

Setting a value for the `rise_capacitance` attribute requires that a value for the `fall_capacitance` also be set, and setting a value for the `fall_capacitance` requires that a value for the `rise_capacitance` also be set.

Note:

If the Library Compiler tool does not find a `capacitance` attribute for an input or inout pin, it generates a warning message and uses the larger of the value settings for `fall_capacitance` and `rise_capacitance`. If `fall_capacitance` and `rise_capacitance` have not been set for the input pin, the Library Compiler tool uses the value to which the `default_input_pin_cap` attribute is set, and for the inout pin, it uses the value to which the `default_inout_pin_cap` attribute is set.

Syntax

```
rise_capacitance : float ;  
  
float
```

A floating-point number in units consistent with other capacitance specifications throughout the library. Typical units of measure for `rise_capacitance` include picofarads and standardized loads.

Example

The following example defines the A and B pins in an AND cell, each with a `fall_capacitance` of one unit, a `rise_capacitance` of two units, and a capacitance of two units.

```
cell (AND) {  
    area : 3 ;  
    pin (A,B) {  
        direction : input ;  
        fall_capacitance : 1 ;  
        rise_capacitance : 2 ;  
        capacitance : 2 ;  
    }  
}
```

steady_state_resistance Attributes

When there are multiple drivers connected to an interconnect network driven by library cells and there is no direct current path between them, the driver resistances could take on different values.

Use the following attributes for more-accurate modeling of steady state driver resistances in library cells.

- `steady_state_resistance_above_high`
- `steady_state_resistance_below_low`
- `steady_state_resistance_high`
- `steady_state_resistance_low`

Example

```
steady_state_resistance_above_high : 200 ;
```

test_output_only Attribute

This attribute is an optional Boolean attribute that you can set for any output port described in statetable format.

In ff or latch format, if a port is to be used for both function and test, you provide the functional description using the `function` attribute. If a port is to be used for test only, you omit the `function` attribute.

Regardless of ff or latch statetable, the `test_output_only` attribute takes precedence over the functionality.

In statetable format, however, a port always has a functional description. Therefore, if you want to specify that a port is for test only, you set the `test_output_only` attribute to true.

Example

```
pin (my_out) {  
    direction : output ;  
    signal_type : test_scan_out ;  
    test_output_only : true ;  
}
```

Describing Design Rule Checks

To define design rule checks, use the following `pin` group attributes and group:

- `fanout_load` attribute
- `max_fanout` attribute
- `min_fanout` attribute
- `max_transition` attribute
- `max_trans` group

- `min_transition` attribute
- `max_capacitance` attribute
- `max_cap` group
- `min_capacitance` attribute
- `cell_degradation` group

fanout_load Attribute

The `fanout_load` attribute gives the fanout load value for an input pin.

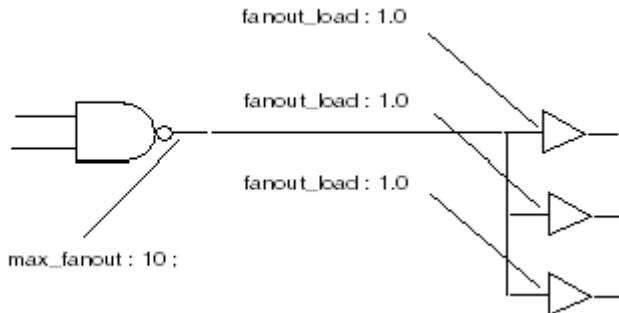
Example

```
fanout_load : 1.0;
```

The sum of all `fanout_load` attribute values for input pins connected to a driving (output) pin must not exceed the `max_fanout` value for that output pin.

[Figure 7-3](#) illustrates `max_fanout` and `fanout_load` attributes for a cell.

Figure 7-3 Fanout Attributes



max_fanout Attribute

This attribute defines the maximum fanout load that an output pin can drive.

Example

```
pin(Q)
  direction : output;
  max_fanout : 10;
}
```

Some designs have limitations on input load that an output pin can drive regardless of any loading contributed by interconnect metal layers. To limit the number of inputs on the same net driven by an output pin, define a `max_fanout` value for each output pin and a `fanout_load` on each input pin in a cell. (See [Figure 7-3](#).)

You can specify `max_fanout` at three levels: at the library level, at the pin level, and on the command line in `dc_shell`.

The `max_fanout` attribute is an implied design rule constraint. The Design Compiler tool attempts to resolve `max_fanout` violations, possibly at the expense of other design constraints.

In [Figure 7-3](#), the Design Compiler tool adds all the `fanout_load` values of the input to equal 3.0. Then the Design Compiler tool compares the sum of the `fanout_load` on all inputs with the `max_fanout` constraint of the driving output, which is 10. In this case, no design rule violation occurs. If a design rule violation occurs, the Design Compiler tool might replace the driving cell with a cell that has a higher `max_fanout` value.

When you are selecting a `max_fanout` value, the Design Compiler tool follows these rules:

- The value you assign for the pin always overrides the library default.
- When you assign a `max_fanout` value at the command line, the Design Compiler tool uses the smallest value. If the pin value is smaller, the Design Compiler tool ignores the command-line value.

To determine `max_fanout`, find the smallest loading of any input to a cell in the library and use that value as the standard load unit for the entire library. Usually the smallest buffer or inverter has the lowest input pin loading value. Use some multiple of the standard value for the fanout loads of the other cells. The Library Compiler tool supports both real and integer loading values.

To specify a global maximum fanout for all gate outputs in the design, use `max_fanout` on the command line of `dc_shell`. When defined on the command line, the value of `max_fanout` is assigned to all gate outputs that do not have a specified `max_fanout` or whose `max_fanout` value is greater than the one defined.

Although you can use capacitance as the unit for your `max_fanout` and `fanout_load` specifications, it should be used to constrain routability requirements.

It differs from the capacitance pin attribute in the following ways:

- The fanout values are used only for design rule checking.
- The capacitance values are used by the Design Compiler timing verifier in delay calculations.

min_fanout Attribute

This attribute defines the minimum fanout load that an output or inout pin can drive. The sum of fanout cannot be less than the minimum fanout value.

Example

```
pin(Q) {
    direction : output;
    min_fanout : 2.0;
}
```

The `min_fanout` attribute is an implied design rule constraint. The Design Compiler tool resolves `min_fanout` violations, possibly at the expense of other design constraints.

max_transition Attribute

This attribute defines a design rule constraint for the maximum acceptable transition time of an input or output pin.

Example

```
pin(A) {
    direction : input;
    max_transition : 4.2;
}
```

You can specify `max_transition` at three levels: at the library level, at the pin level, and on the command line in `dc_shell`.

With an output pin, `max_transition` is used only to drive a net for which the cell can provide a transition time at least as fast as the defined limit.

With an input pin, `max_transition` indicates that the pin cannot be connected to a net that has a transition time greater than the defined limit. The Design Compiler tool attempts to resolve `max_transition` violations, possibly at the expense of other design constraints.

In the following example, the cell that contains pin Q cannot be used to drive a net for which the cell cannot provide a transition time faster than 5.2:

```
pin(Q) {
    direction : output ;
    max_transition : 5.2 ;
}
```

The `max_transition` value you define is checked by the synthesis tool, the transition delay that the Design Compiler tool calculates is the rise and fall resistance multiplied by the sum of the pin and wire capacitances.

If the calculated delay is greater than the value you specify with the `max_transition` attribute, a design rule violation is reported, and the Design Compiler tool tries to correct the violation, possibly at the expense of other design constraints.

max_trans Group

The `max_trans` group specifies the maximum transition time of an input, output, or inout pin as a function of operating frequency of the cell, input transition time, and output load. Use the `max_trans` group instead of the `max_transition` attribute to include these effects on the maximum transition time. When both the `max_trans` group and `max_transition` attribute are present, the `max_trans` group overrides the `max_transition` attribute.

To model the effect of operating frequency, input transition time, and output load on the maximum transition, define the lookup table template for the `max_trans` group at the library level.

To define the lookup table template, use the `maxtrans_lut_template` group at the library level. The `maxtrans_lut_template` group can have the variables, `variable_1`, `variable_2`, and `variable_3`. The valid values of the `variable_1`, `variable_2`, and `variable_3` variables are `frequency`, `input_transition_time`, and `total_output_net_capacitance` respectively.

The one-dimensional lookup table consists of the maximum transition values for different values of `frequency`. Similarly, the two-dimensional lookup table consists of the maximum transition values for different values of `frequency` and `input_transition_time` and so on.

The following syntax shows the maximum transition model:

```
library (library_name) {
    delay_model : table_lookup;
    ...
    maxtrans_lut_template (template_name1) { /*1-D LUT template*/
        variable_1 : frequency ;
        index_1 ( "float, ... float" );
    }

    maxtrans_lut_template (template_name2) { /*2-D LUT template*/
        variable_1 : frequency ;
        variable_2 : input_transition_time ;
        index_1 ( "float, ... float" );
        index_2 ( "float, ... float" );
    }

    maxtrans_lut_template (template_name3) { /*3-D LUT template*/
        variable_1 : frequency ;
        variable_2 : input_transition_time ;
        variable_3 : total_output_net_capacitance ;
        index_1 ( "float, ... float" );
        index_2 ( "float, ... float" );
        index_3 ( "float, ... float" );
    }
    cell (cell_name) {
        ...
        pin (pin_name) { /* input pin */
            ...
        }
    }
}
```

```

max_trans (template_name1) {
    index_1 ("float, ... float");
    values ("float, ... float");
}
...
} /* pin */
...
pin (pin_name) { /* input pin */
...
max_trans (template_name2) { /* output or inout pin */
    index_1 ("float, ... float");
    index_2 ("float, ... float");
    values ("float, ... float");
    values ("float, ... float");
}
...
pin (pin_name2) { /* input pin */
...
max_trans (template_name3) { /* output or inout pin */
    index_1 ("float, ... float");
    index_2 ("float, ... float");
    index_3 ("float, ... float");
    values ("float, ... float");
    values ("float, ... float");
    values ("float, ... float");
}
}
} /* pin */
}/* cell */
...
}/* library */

```

Maximum Transition Modeling Example

[Example 7-7](#) shows a frequency-dependent maximum transition model with a one-dimensional lookup table.

Example 7-7 Frequency-Dependent Maximum Transition Model Using Lookup Table

```

library(my_lib) {
    technology (cmos);
    delay_model : table_lookup;
    ...
    maxtrans_lut_template ( mt ) {
        variable_1: frequency;
        index_1 ( "100.0000, 200.0000" );
    }
    ...
    cell ( test ) {
        .....
        pin(Z) {
            direction : output ;
            function: "A";
        }
    }
}

```

```

max_trans(mt) {
    index_1 ( "100.0000, 200.0000, 300.0000");
    values ( "925.0000, 835.0000, 745.0000");
} /* end of max_trans */
timing () {
    related_pin : "A" ;
    ...
} /* end of arc */
} /* end of pin Z */
pin(A) {
    direction : input ;
    max_transition : 5000.000000 ;
    capacitance : 1.0 ;
    } /* end of pin A */
} /* end of cell */
} /* end of library */

```

max_capacitance Attribute

This attribute defines the maximum total capacitive load that an output pin can drive. This attribute can be specified only for an output or inout pin.

Example

```

pin(Q) {
    direction : output;
    max_capacitance : 5.0;
}

```

You can specify `max_capacitance` at three levels: at the library level, at the pin level, and on the command line in `dc_shell`.

The Design Compiler tool uses an output pin only when the pin has a `max_capacitance` attribute value greater than or equal to the total wire and pin capacitive load it drives. The total wire and pin capacitance is derated before it is checked by this tool. The tool attempts to resolve `max_capacitance` design rule violations, possibly at the expense of other design constraints.

max_cap Group

The `max_cap` group specifies the maximum capacitive load that an output or inout pin can drive as a function of operating frequency of the cell or both operating frequency and input transition time. Use the `max_cap` group instead of the `max_capacitance` attribute to include these effects on the maximum capacitance. When both the `max_cap` group and `max_capacitance` attribute are present, the `max_cap` group overrides the `max_capacitance` attribute.

To model the effect of operating frequency and input transition time on the maximum capacitance, define the lookup table template for the `max_cap` group at the library level.

To define the lookup table template, use the `maxcap_lut_template` group at the library level. The `maxcap_lut_template` group can have the variables, `variable_1` and `variable_2`. The valid values of the `variable_1` and `variable_2` variables are `frequency` and `input_transition_time`, respectively.

The one-dimensional lookup table consists of the maximum capacitance values for different values of `frequency`. The `max_cap` group takes the value of the maximum capacitance from the lookup table by using the `variable_1` variable.

The two-dimensional lookup table consists of the maximum capacitance values for different values of `frequency` and `input_transition_time`. The `max_cap` group takes the value of the maximum capacitance from the lookup table by using the `variable_1` and `variable_2` variables.

The following shows the maximum capacitance model syntax:

```
library (library_name) {
    delay_model : table_lookup;
    ...
    maxcap_lut_template (template_name1) /*1-D lookup table template*/
        variable_1 : frequency ;
        index_1 ("float, ... float");
    }
    maxcap_lut_template (template_name2) /*2-D LUT template */
        variable_1 : frequency ;
        variable_2 : input_transition_time ;
        index_1 ("float, ... float");
        index_2 ("float, ... float");
    }
    cell (cell_name) {
        ...
        pin (pin_name) {
            ...
            max_cap (template_name1) {
                index_1 ("float, ... float");
                values ("float, ... float");
            }
            ...
        } /* pin */
        ...
    }/* cell */
    ...
}/* library */
```

Maximum Capacitance Modeling Example

[Example 7-8](#) shows a frequency-dependent maximum capacitance model with a one-dimensional lookup table.

Example 7-8 Frequency-Dependent Maximum Capacitance Model Using Lookup Table

```
library(example_library) {
  technology (cmos);
  delay_model : table_lookup;
  ...
  maxcap_lut_template ( mc ) {
    variable_1 : frequency;
    index_1 ( "100.0000, 200.0000" );
  }
  ...
  cell ( test ) {
    .....
    pin(Z) {
      direction : output ;
      function: "A";
      max_transition : 5000.000000 ;
      min_transition : 0.000000 ;
      min_capacitance : 0.000000 ;
      max_cap(mc) {
        index_1 ( "100.0000, 200.0000, 300.0000");
        values ( "925.0000, 835.0000, 745.0000");
      } /* end of max_cap */
      timing () {
        related_pin : "A" ;
        ...
      } /* end of arc */
    } /* end of pin Z */
    pin(A) {
      direction : input ;
      max_transition : 5000.000000 ;
      capacitance : 1.0 ;
    } /* end of pin A */
  } /* end of cell */
} /* end of library */
```

min_capacitance Attribute

This attribute defines the minimum total capacitive load that an output pin can drive. The capacitance load cannot be less than the minimum capacitance value. This attribute can be specified only for an output or inout pin.

Example

```
pin(Q) {
  direction : output;
  min_capacitance : 1.0;
}
```

The Design Compiler tool uses an output pin only when the pin has a `min_capacitance` attribute value less than or equal to the total wire and pin capacitive load it drives. The total wire and pin capacitance is derated before it is checked by the tool. The tool attempts to

`resolve min_capacitance` design rule violations, possibly at the expense of other design constraints.

cell_degradation Group

Use the `cell_degradation` group to describe a cell performance degradation design rule when compiling a design. A cell degradation design rule specifies the maximum capacitive load a cell can drive without causing cell performance degradation during the fall transition.

This description is restricted to functionally related input and output pairs. You can determine the degradation value by switching some inputs while keeping other inputs constant. This causes output discharge. The degradation value for a specified input transition rate is the maximum output loading that does not cause cell degradation.

You can model cell degradation only in libraries using the CMOS nonlinear delay model. Cell degradation modeling uses the same format of templates and lookup tables used to model delay with the nonlinear delay model.

There are two ways to model cell degradation,

1. Create a one-dimensional lookup table template that is indexed by input transition time.
1. The following shows the syntax of the cell degradation template.

```
lu_table_template(template_name) {
    variable_1 : input_net_transition;
    index_1 ("float, ..., float");
}
```

The valid value for `variable_1` is `input_net_transition`.

The `index_1` values must be greater than or equal to 0.0 and follow the same rules for the lookup table template `index_1` attribute described in “[Defining pin Groups](#)” on [page 7-17](#). The number of floating-point numbers in `index_1` determines the size of the table dimension.

This is an example of a cell degradation template.

```
lu_table_template(deg_constraint) {
    variable_1 : input_net_transition;
    index_1 ("0.0, 1.0, 2.0");
}
```

See [Chapter 11, “Timing Arcs”](#), for more information about lookup table templates.

2. Use the `cell_degradation` group and the cell degradation template to create a one-dimensional lookup table for each timing arc in the cell. You receive warning messages if you define a `cell_degradation` construct for some, but not all, timing arcs in the cell.

The following example shows the `cell_degradation` group:

```

pin(output) {
    timing() {
        cell_degradation(deg_constraint) {
            index_1 ("0.5, 1.5, 2.5");
            values ("0.0, 2.0, 4.0");
        }
    }
}

```

3. You can describe cell degradation groups only in the following types of `timing` groups:
- combinational
 - `three_state_enable`
 - `rising_edge`
 - `falling_edge`
 - `preset`
 - `clear`

Assigning Values to Lookup Tables

These are the rules for specifying cell degradation lookup tables:

- You can overwrite the `index_1` value in the lookup table template with the optional `index_1` attribute in the `cell_degradation` group, but the overwrite must occur before the actual values are defined.
- The number of floating-point numbers in the `value` attribute must be the same as the number in the corresponding `index_1` in the table template.
- Each entry in the `values` attribute uses the `capacitive_load_unit` library attribute as its unit and must be 0.0 or greater.
- When a timing arc is state-dependent and the cell has cell degradation defined for all valid timing types, you must create a cell degradation table for each state-dependent timing arc.

Describing Clocks

To define clocks and clocking, use these `pin` group attributes:

- `clock`
- `min_period`

- `min_pulse_width_high`
- `min_pulse_width_low`

clock Attribute

This attribute indicates whether or not an input pin is a clock pin.

A true value labels a pin as a clock pin. A false value labels a pin as not a clock pin, even though it might otherwise have such characteristics.

Example

```
clock : true ;
```

min_period Attribute

Place the `min_period` attribute on the clock pin of a flip-flop or a latch to specify the minimum clock period required for the input pin. The minimum period is the sum of the data arrival time and setup time. This time must be consistent with the `max_transition` time.

Example

```
min_period : 26.0;
```

If a `min_period` attribute is placed on a pin that is not a clock pin, the Library Compiler tool ignores the attribute.

min_pulse_width_high and min_pulse_width_low Attributes

Use these optional attributes to specify the minimum length of time a pin must remain at logic 1 (`min_pulse_width_high`) or logic 0 (`min_pulse_width_low`). These attributes can be placed on a clock input pin or an asynchronous clear/preset pin of a flip-flop or latch.

Note:

The Design Compiler tool does not support minimum pulse width as a constraint during synthesis.

Example

The following example shows both attributes on a clock pin, indicating the minimum pulse width for a clock pin.

```
pin(CLK) {
    direction : input ;
    capacitance : 1 ;
    min_pulse_width_high : 3 ;
    min_pulse_width_low : 3 ;
}
```

Describing Clock Pin Functions

To define the function of a clock pin, use these `pin` group attributes:

- `function`
- `three_state`
- `x_function`
- `state_function`
- `internal_node`

function Attribute

The `function` attribute defines the value of an output or inout pin in terms of the cell's input or inout pins.

Syntax

```
function : "Boolean expression" ;
```

The precedence of the operators is left to right, with inversion performed first, then XOR, then AND, then OR.

Table 7-6 lists the Boolean operators that are valid in a `function` statement.

Table 7-6 Valid Boolean Operators

Operator	Description
,	Invert previous expression
!	Invert following expression
^	Logical XOR
*	Logical AND
&	Logical AND
space	Logical AND
+	Logical OR
	Logical OR
1	Signal tied to logic 1
0	Signal tied to logic 0

The `function` attribute statement provides information for the Design Compiler and DFT Compiler tools. The Design Compiler tool uses this information when choosing components during synthesis. A cell without a `function` attribute is treated as a black box and is not synthesized or optimized.

If you want to prevent a cell from being used or replaced during optimization, use the `dont_use` attribute.

Note:

The DFT Compiler tool cannot test cells that do not include a `function` attribute or cells that cannot have a `function` attribute, such as shift registers and counters.

Grouped Pins in function Statements

Grouped pins can be used as variables in a `function` statement; see “[Defining Bused Pins](#)” on page 7-58 and “[Defining Signal Bundles](#)” on page 7-65. In `function` statements that use bus or bundle names, all the variables in the statements must be either a single pin or buses or bundles of the same width.

Ranges of buses or bundles are valid if the range you define contains the same number of members as the other buses or bundles in the same expression. You can reverse the bus

order by listing the member numbers in reverse (high: low) order. Two buses, bundles, or bused-pin ranges with different widths should not appear in the same `function` statement; otherwise, the Library Compiler tool generates an error message.

When the `function` attribute of a cell with group input pins is a combinational-logic function of grouped variables only, the logic function is expanded to apply to each set of output grouped pins independently. For example, if A, B, and Z are defined as buses of the same width and the function statement for output Z is

```
function : "(A & B)" ;
```

the function for Z[0] is interpreted as

```
function : "(A[0] & B[0])" ;
```

Likewise, the function for Z[1] is interpreted as

```
function : "(A[1] & B[1])" ;
```

If a bus and a single pin are in the same `function` attribute, the single pin is distributed across all members of the bus. For example, if A and Z are buses of the same width, B is a single pin, and the function statement for the Z output is

```
function : "(A & B)" ;
```

The function for Z[0] is interpreted as

```
function : "(A[0] & B)" ;
```

Likewise, the function for Z[1] is interpreted as

```
function : "(A[1] & B)" ;
```

three_state Attribute

Use this attribute to define a three-state output pin in a cell.

Only the high impedance to logic 0 and high impedance to logic 1 timing arcs are used during three-state component synthesis and optimization.

In [Example 7-9](#), output pin Z is a three-state output pin. When input pin E (enable) goes low, pin Z goes to a high-impedance state. The value of the `three_state` attribute is, therefore, E'.

Example 7-9 Three-State Cell Description

```
library(example) {
    technology (cmos) ;
    date : "May 14, 2002" ;
    revision : 2002.05;
    ...
    cell(TRI_INV2) {
        area : 3 ;
```

```

pin(A) {
    direction : input ;
    capacitance : 2 ;
}
pin(E) {
    direction : input ;
    capacitance : 2 ;
}
pin(Z) {
    direction : output ;
    function : "A'" ;
    three_state : "E'" ;
    timing() {
        ...
    }
}
}

```

x_function Attribute

Use the `x_function` attribute to describe the X behavior of a pin, where X is a state other than 0, 1, or Z.

Note:

Only the Formality tool uses the `x_function` attribute.

The `three_state`, `function`, and `x_function` attributes are defined for output and inout pins and can have shared input. You can assign `three_state`, `function`, and `x_function` to be the function of the same input pins. When these functions have shared input, however, the cell is not inserted by the Design Compiler tool. It must be inserted manually.

Also, when the values of more than one function equal 1, the three functions are evaluated in this order:

1. `x_function`
2. `three_state`
3. `function`

Example

```

pin (y) {
    direction: output;
    function : "!ap * !an" ;
    x_function : "!ap * an" ;
    three_state : "ap * !an" ;
}

```

state_function Attribute

Use this attribute to define output logic. Ports in the `state_function` Boolean expression must be either input, three-state inout, or ports with an `internal_node` attribute. If the output logic is a function of only the inputs (IN), the output is purely combinational (for example, feed-through output). A port in the `state_function` expression refers only to the non-three-state functional behavior of that port. An inout port in the `state_function` expression is treated only as an input port.

Example

```
state_function : Q*X;
```

internal_node Attribute

Use this attribute to resolve node names to real port names. The `internal_node` attribute describes the sequential behavior of an output pin. It provides the relationship between the `statetable` group and a pin of a cell. Each output with the `internal_node` attribute might also have the optional `input_map` attribute.

Example

```
internal_node : "Q";
```

Describing Sequential Devices

To describe an input pin of a flip-flop or latch, use the `prefer_tied` attribute.

prefer_tied Attribute

When it is processing each sequentially modeled component, the Library Compiler tool generates the combinational logic necessary to implement the D flip-flop functionality that uses that component. Because D flip-flop functionality can be implemented in more than one way by use of a complex flip-flop, the `prefer_tied` pin attribute lets you choose the logic that results in tying the pin to a specified logic value.

Example

```
prefer_tied : "1";
```

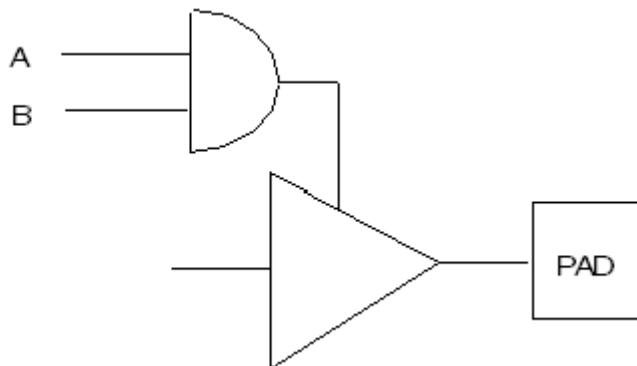
The Library Compiler tool considers as many `prefer_tied` attributes as possible while it is still able to implement D flip-flop functionality. However, it cannot honor all of them. For example, if the library developer specifies `prefer_tied : 0` on all the inputs, the Library Compiler tool honors as many as possible and ignores the rest. If the Library Compiler tool ignores any inputs, it issues a message stating so when reading in the `.lib` file. You typically use `prefer_tied` on multiplexed flip-flop input.

Note:

The `prefer_tied` attribute directs the Design Compiler tool in implementing a D flip-flop. It is important to define all complex register elements with the `prefer_tied` attribute. For sequential optimization, the Design Compiler tool ignores this attribute.

The `prefer_tied` attribute can also be used to indicate which pins are tied to fixed logic during three_state degeneration for three-state pad cells. For example, the Library Compiler tool tries to degenerate the input pad shown in [Figure 7-4](#) into a simple three-state output pad by tying either pin A or pin B to logic 1. The Design Compiler tool uses the degenerated pad cell for pad-related operations.

Figure 7-4 Output Pad Tied to Fixed Logic



To control the degeneration process to prefer one version over the other, place a `prefer_tied` attribute on the pin. the Library Compiler tool ties that pin to the logic specified in the attribute.

If the Library Compiler tool cannot degenerate the pad cell according to the specified `prefer_tied` attribute but can degenerate the cell by tying the same pin to the opposite logic, it does so and issues a warning.

In this example, to tie pin B rather than pin A to logic 1 during degeneration, specify a `prefer_tied` attribute in the `pin` group of pin B.

```

pin (B) {
    prefer_tied : "1";
    ...
}
  
```

Describing Pad Pins

To describe a pad pin, use these `pin` group attributes:

- `is_pad`
- `multicell_pad_pin`

- input_voltage
- output_voltage
- pulling_resistance
- pulling_current
- hysteresis
- drive_current
- slew_control
- rise_current_slope_before_threshold
- rise_current_slope_after_threshold
- fall_current_slope_before_threshold
- fall_current_slope_after_threshold
- rise_time_before_threshold
- rise_time_after_threshold
- fall_time_before_threshold
- fall_time_after_threshold

is_pad Attribute

After you identify a cell as a pad cell, you must indicate which pin represents the pad. The valid values are `true` and `false`. The `is_pad` attribute must be used on at least one pin with `a_pad_cell` attribute. You can also specify the `is_pad` attribute on a PG pin. If the `pad_cell` attribute is specified on a I/O cell, you must set the `is_pad` attribute to `true` in either a `pg_pin` group or on a signal pin for that cell. Otherwise, the Library Compiler tool issues an error message. If the cell-level `pad_cell` attribute is specified on a I/O cell and there is no signal pin or PG pin in the pad cell, Library Complier issues a warning message. For information about the `pad_cell` attribute, see “[pad_cell Attribute](#)” on page 7-9.

Examples

```
cell(INBUF) {  
    ...  
    pin(PAD) {  
        direction : input;  
        is_pad : true;  
    }  
}
```

In the following example, the `is_pad` attribute is specified on a PG pin:

```
cell (POWER_PAD_CELL) {
    ...
    pad_cell : true ;
    pg_pin (my_pg_pin) {
        is_pad : true ;
        ...
    }
    pin (my_pin) {
        ...
    }
}
```

multicell_pad_pin Attribute

Use this attribute to indicate which pin on a cell should be connected to another cell to create the correct configuration. For example, some technologies implement pads with multiple cells by connecting a pad cell and a pad driver.

Example

```
multicell_pad_pin : true;
```

Use this attribute for all pins on a pad or auxiliary pad cell that are connected to another cell. The Design Compiler tool connects the correct multicell pad pins according to the `connection_class` attribute.

Voltage Attributes

These are voltage attributes:

- `input_voltage` is used for an input pin definition.
- `output_voltage` is used for an output pin definition.

You can define a special set of voltage thresholds in the `library` group with the `input_voltage` or `output_voltage` attribute. You can then apply the default voltage ranges in the group to selected cells by using the `input_voltage` or `output_voltage` attribute in the pin definition.

[Example 7-10](#) selects a voltage range for the input pin of a pad cell.

Example 7-10 Voltage Range for Input Pin

```
input_voltage (CMOS_SCHMITT) {
    ...
}
cell (INBUF) {
    ...
    pin (PAD) {
        input_voltage : CMOS_SCHMITT ;
```

```
    ...
}
```

Pull-Up and Pull-Down Strength Attributes

The pull-up and pull-down strength attributes are

- `pulling_resistance`
- `pulling_current`

These attributes let you define the resistance or current-drawing capability of a pull-up or pull-down device on a pin. These attributes can be used for pins with a driver type of `pull_up` or `pull_down`.

Example

```
driver_type : pull_up ;
pulling_resistance : 1000 ;
```

Assigning a negative value to the `pulling_current` attribute produces a current flow in the opposite or negative direction. Assigning a negative value to the `pulling_resistance` attribute produces a warning and converts that value to 0.

hysteresis Attribute

Use the `hysteresis` attribute on an input pad when you anticipate a long transition time or when you expect the pad to be driven by a particularly noisy line.

Example

```
hysteresis : true;
```

This attribute allows the pad to accommodate longer transition times, which are more subject to noise problems. A pad with this option has wider threshold tolerances, so voltage spikes on the input do not propagate to the core cells. Using the `hysteresis` attribute cuts down on power consumption, because the input signals change only after the threshold is exceeded.

The derating factors of pads with `hysteresis` can sometimes be different from those of core cells.

Drive-Current Strength Attribute

Most output and bidirectional pad cells are characterized with an approximate drive-current capability. You select the pad with the lowest drive current that meets your alternating current's timing constraints.

Example

```
drive_current : 5.0;
```

Each library cell can have only one `drive_current` attribute. You might need to create a different pad cell for each drive-current possibility in your library. Output and inout pads must have the `drive_current` attribute assigned.

Slew-Rate Control Attributes

Slew-rate control limits peak noise by smoothing out fast output transitions, thus decreasing the possibility of a momentary disruption in the power or ground planes.

The Library Compiler tool lets you describe two levels of slew-rate control, using the `slew_control` attribute for coarse tuning and the threshold attributes for fine tuning.

slew_control Attribute

The more slew-rate control you apply, the slower the output transitions are. The `slew_control` attribute provides increasing levels of slew-rate control to slow down the transition rate.

Threshold Attributes

You can define slew-rate control in terms of the current versus time characteristics of the output pad (di/dT). The syntax is

```
threshold_attribute : value ;
```

The `value` is a floating-point number in the units specified by `current_unit` for current-related attributes and `time_unit` for time-related attributes.

The eight threshold attributes are

`rise_current_slope_before_threshold`

This value represents a linear approximation of the change in current with respect to time from the beginning of the rising transition to the threshold point.

`rise_current_slope_after_threshold`

This value represents a linear approximation of the change in current with respect to time from the point at which the rising transition reaches the threshold to the end of the transition.

`fall_current_slope_before_threshold`

This value represents a linear approximation of the change in current with respect to time from the beginning of the falling transition to the threshold point.

fall_current_slope_after_threshold

This value represents a linear approximation of the change in current with respect to time from the point at which the falling transition reaches the threshold to the end of the transition.

rise_time_before_threshold

This value gives the time interval from the beginning of the rising transition to the point at which the threshold is reached.

rise_time_after_threshold

This value gives the time interval from the threshold point of the rising transition to the end of the transition.

fall_time_before_threshold

This value gives the time interval from the beginning of the falling transition to the point at which the threshold is reached.

fall_time_after_threshold

This value gives the time interval from the threshold point of the falling transition to the end of the transition.

[Example 7-11](#) shows the slew-rate control attributes on an output pad pin.

Example 7-11 Slew-Rate Control Attributes

```
pin(PAD) {  
    is_pad : true;  
    direction : output;  
    output_voltage : GENERAL;  
    slew_control : high;  
    rise_current_slope_before_threshold : 0.18;  
    rise_time_before_threshold : 0.8;  
    rise_current_slope_after_threshold : -0.09;  
    rise_time_after_threshold : 2.4;  
    fall_current_slope_before_threshold : -0.14;  
    fall_time_before_threshold : 0.55;  
    fall_current_slope_after_threshold : 0.07;  
    fall_time_after_threshold : 1.8;  
    ...  
}
```

Defining Bused Pins

To define bused pins, use these groups:

- `type group`
- `bus group`

You can use a defined bus or bus member in Boolean expressions in the `function` attribute. An output pin does not need to be defined in a cell before it is referenced.

type Group

If your library contains bused pins, you must define `type` groups and define the structural constraints of each bus type in the library.

The `type` group is defined at the `library` group level, as follows:

```
library (lib_name) {  
    type ( name ) {  
        ... type description ...  
    }  
}
```

name

Identifies the bus type.

A `type` group can one of the following:

base_type

Only the array base type is supported.

data_type

Only the bit data type is supported.

bit_width

An integer that designates the number of bus members. The default is 1.

bit_from

An integer indicating the member number assigned to the most significant bit (MSB) of successive array members. The default is 0.

bit_to

An integer indicating the member number assigned to the least significant bit (LSB) of successive array members. The default is 0.

downto

A value of true indicates that member number assignment is from high to low instead of low to high. The default is false (low to high).

[Example 7-12](#) illustrates a `type` group statement.

Example 7-12 type Group Statement

```
type ( BUS4 ) {
    base_type : array ;
    data_type : bit ;
    bit_width : 4 ;
    bit_from : 0 ;
    bit_to : 3 ;
    downto :false ;
}
```

It is not necessary to use all the `type` group attributes. For example, the `type` group statements in [Example 7-13](#) are both valid descriptions of BUS4 in [Example 7-12](#).

Example 7-13 Alternative type Group Statements

```
type ( BUS4 ) {
    base_type : array ;
    data_type : bit ;
    bit_width : 4 ;
    bit_from : 0 ;
    bit_to : 3 ;
}
type ( BUS4 ) {
    base_type : array ;
    data_type : bit ;
    bit_width : 4 ;
    bit_from : 3 ;
    downto : true ;
}
```

Because the Library Compiler tool checks the attributes you use for consistency, using all of them is a good way to verify your description.

After you define a `type` group, you can use the `type` group in a `bus` group to describe bused pins.

bus Group

A `bus` group describes the characteristics of a bus. You define it in a `cell` group, as shown here:

```
library (lib_name) {
    cell (cell_name) {
        area : float ;
        bus ( name ) {
            ... bus description ...
        }
    }
}
```

A `bus` group contains the following elements:

- `bus_type` attribute
- pin groups

In a `bus` group, use the number of bus members (pins) defined by the `bit_width` attribute in the applicable `type` group. You must declare the `bus_type` attribute first in the `bus` group.

bus_type Attribute

The `bus_type` attribute specifies the bus type. It is a required element of all `bus` groups. Always declare the `bus_type` as the first attribute in a `bus` group.

Note:

The bus type name must exist in a `type` group.

Syntax

```
bus_type : name ;
```

Pin Attributes and Groups

Pin attributes in a `bus` or `bundle` group specify default attribute values for all pins in that bus or bundle. Pin attributes can also appear in pin groups inside the `bus` or `bundle` group to define attribute values for specific bus or bundle pins or groups of pins. Values used in pin groups override the default attribute values defined for the bus or bundle.

All pin attributes are valid inside `bus` and `pin` groups. See “[General pin Group Attributes](#)” on [page 7-19](#) for a description of pin attributes. The `direction` attribute value of all bus members must be the same.

Use the full name of a pin for the names of pins in a `pin` group contained in a `bus` group.

The following example shows a `bus` group that defines bus A, with defaults for direction and capacitance assigned:

```
bus (A) {
    bus_type : bus1 ;
    direction : input ;
    capacitance : 3 ;
    ...
}
```

The following example illustrates a `pin` group that defines a new `capacitance` attribute value for pin 0 in bus A:

```
pin (A[0]) {
    capacitance : 4 ;
}
```

You can also define pin groups for a range of bus members. A range of bus members is defined by a beginning value and an ending value, separated by a colon. No spaces can appear between the colon and the member numbers.

The following example illustrates a `pin` group that defines a new `capacitance` attribute value for bus members 0, 1, 2, and 3 in bus A:

```
pin (A[0:3]) {
    capacitance : 4 ;
}
```

For nonbused pins, you can identify member numbers as single numbers or as a range of numbers separated by a colon. Do not define member numbers in a list.

See [Table 7-7](#) for a comparison of bused and single-pin formats.

Table 7-7 Comparison of Bused and Single-Pin Formats

Pin type	Technology library
Bused Pin	pin x[3:0]
Single Pin	pin x

Example Bus Description

[Example 7-14](#) is a complete bus description that includes `type` and `bus` groups. It also shows the use of bus variables in `function`, `related_pin`, `pin_opposite`, and `pin_equal` attributes.

Example 7-14 Bus Description

```
library (ExamBus) {
```

```
date : "May 14, 2002";
revision : 2002.05;
bus_naming_style :"%s[%d]";/* Optional; this is the
                           default */
type (bus4) {
    base_type : array; /* Required */
    data_type : bit; /* Required if base_type is array */
    bit_width : 4; /* Optional; default is 1 */
    bit_from : 0; /* Optional MSB; defaults to 0 */
    bit_to : 3; /* Optional LSB; defaults to 0 */
    downto : false; /* Optional; defaults to false */
}
cell (bused_cell) {
    area : 10;
    bus (A) {
        bus_type : bus4;
        direction : input;
        capacitance : 3;
        pin (A[0:2]) {
            capacitance : 2;
        }
        pin (A[3]) {
            capacitance : 2.5;
        }
    }
    bus (B) {
        bus_type : bus4;
        direction : input;
        capacitance : 2;
    }
    pin (E) {
        direction : input ;
        capacitance 2 ;
    }
    bus(X) {
        bus_type : bus4;
        direction : output;
        capacitance : 1;
        pin (X[0:3]) {
            function : "A & B'";
            timing() {
                related_pin : "A B";
                /* A[0] and B[0] are related to X[0],
                   A[1] and B[1] are related to X[1], etc. */
            }
        }
    }
    bus (Y) {
        bus_type : bus4;
        direction : output;
        capacitance : 1;
        pin (Y[0:3]) {
            function : "B";
        }
    }
}
```

```
    three_state : "!E";
    timing () {
        related_pin : "A[0:3] B E";
    }
    internal_power() {
        when: "E" ;
        related_pin : B ;
        power() {
            ...
        }
    }
    internal_power() {
        related_pin : B ;
        power() {
            ...
        }
    }
}
bus (Z) {
    bus_type : bus4;
    direction : output;
    pin (Z[0:1]) {
        function : "!A[0:1]";
        timing () {
            related_pin : "A[0:1]";
        }
        internal_power() {
            related_pin : "A[0:1]";
            power() {
                ...
            }
        }
    }
    pin (Z[2]) {
        function "A[2]";
        timing () {
            related_pin : "A[2]";
        }
        internal_power() {
            related_pin : "A[0:1]";
            power() {
                ...
            }
        }
    }
    pin (Z[3]) {
        function : "!A[3]";
        timing () {
            related_pin : "A[3]";
        }
        internal_power() {
            related_pin : "A[0:1]";
        }
    }
}
```

```
power() {
    ...
}
}

pin_opposite("Y[0:1]", "Z[0:1]");
/* Y[0] is opposite to Z[0], etc. */
pin_equal("Y[2:3] Z[2:3]");
/* Y[2], Y[3], Z[2], and Z[3] are equal */

cell (bused_cell2) {
    area : 20;
    bus (A) {
        bus_type : bus41;
        direction : input;
        capacitance : 1;
        pin (A[0:3]) {
            capacitance : 2;
        }
        pin (A[3]) {
            capacitance : 2.5;
        }
    }
    bus (B) {
        bus_type : bus4;
        direction : input;
        capacitance : 2;
    }
    pin (E) {
        direction : input ;
        capacitance 2 ;
    }
    bus(X) {
        bus_type : bus4;
        direction : output;
        capacitance : 1;
        pin (X[0:3]) {
            function : "A & B'";
            timing() {
                related_pin : "A B";
                /* A[0] and B[0] are related to X[0],
                   A[1] and B[1] are related to X[1], etc. */
            }
        }
    }
}
```

Defining Signal Bundles

You need certain attributes to define a bundle. A bundle groups several pins that have similar timing or functionality. The bundle is used only within the Library Compiler tool; the Design Compiler tool cannot refer to a bundle in a netlist. Bundles are used for multibit cells such as multibit latch, multibit flip-flop, and multibit AND gate.

bundle Group

Define a `bundle` group in a `cell` group, as shown:

```
library (lib_name) {
    cell (cell_name) {
        area : float ;
        bundle ( name ) {
            ... bundle description ...
        }
    }
}
```

A `bundle` group contains the following elements:

members attribute

The `members` attribute must be declared first in a `bundle` group.

pin attributes

These include `direction`, `function`, and `three-state`.

members Attribute

The `members` attribute is used in a `bundle` group to list the pin names of the signals in a bundle. The `members` attribute must be included as the first attribute in the `bundle` group. It provides the bundle element names and groups a set of pins that have similar properties. The number of members defines the width of the bundle.

If a bundle has a `function` attribute defined for it, that function is copied to all bundle members. For example,

```
pin (C) {
    direction : input ;
    ...
}
bundle(A) {
    members(A0, A1, A2, A3);
    direction : output ;
    function : "B' + C";
```

```

    ...
}

bundle(B) {
    members(B0, B1, B2, B3);
    direction : input;
    ...
}

```

means that the members of the A bundle have these values:

```

A0 = B0' + C;
A1 = B1' + C;
A2 = B2' + C;
A3 = B3' + C;

```

Each bundle operand (B) must have the same width as the function parent bundle (A).

pin Attributes

For information about pin attributes, see “[General pin Group Attributes](#)” on page 7-19.

[Example 7-15](#) shows a bundle group in a multibit latch.

Example 7-15 Multibit Latch With Signal Bundles

```

cell (latch4) {
    area: 16;
    pin (G) { /* active-high gate enable signal */
        direction : input;
        :
    }
    bundle (D) { /* data input with four member pins */
        members(D1, D2, D3, D4); /*must be first attribute */
        direction : input;
    }
    bundle (Q) {
        members(Q1, Q2, Q3, Q4);
        direction : output;
        function : "IQ" ;
    }
    bundle (QN) {
        members (Q1N, Q2N, Q3N, Q4N);
        direction : output;
        function : "IQN";
    }
    latch_bank(IQ, IQN, 4) {
        enable : "G" ;
        data_in : "D" ;
    }
}

cell (latch5) {
    area: 32;
    pin (G) { /* active-high gate enable signal */

```

```

        direction : input;
        :
    }
bundle (D) { /* data input with four member pins */
    members(D1, D2, D3, D4); /*must be first attribute */
    direction : input;
}
bundle (Q) {
    members(Q1, Q2, Q3, Q4);
    direction : output;
    function : "IQ" ;
}
bundle (QN) {
    members (Q1N, Q2N, Q3N, Q4N);
    direction : output;
    function : "IQN";
}
latch_bank(IQ, IQN, 4) {
    enable : "G" ;
    data_in : "D" ;
}
}
}

```

Defining Layout-Related Multibit Attributes

The `single_bit_degenerate` attribute is a layout-related attribute for multibit cells. The attribute also applies to sequential and combinational cells.

The `single_bit_degenerate` attribute is for use on multibit bundle or bus cells that are black boxes. The value of this attribute is the name of a single-bit library cell, which the Design Compiler tool uses as a link to associate the multibit cell with the single-bit cell in the library.

[Example 7-16](#) shows multibit library cells with the `single_bit_degenerate` attribute.

Example 7-16 Multibit Cells With `single_bit_degenerate` Attribute

```

cell (FDX2) {
    area : 18 ;
    single_bit_degenerate : FDB ;
    bundle (D) {
        members (D0, D1) ;
        direction : input ;
        ...
        timing () {
            ...
            ...
        }
    }
}

cell (FDX4)
area : 18 ;

```

```

single_bit_degenerate : FDB ;
bus (D) {
    bus_type : bus4 ;
    direction : input ;
    ...
    timing () {
        ...
        ...
    }
}
}

```

The Library Compiler tool verifies that all the following conditions exist for cells in a library containing black box cells and using the `single_bit_degenerate` attribute. If any one of the conditions does not exist, the `single_bit_degenerate` attribute is removed from the cell.

- For any `single_bit_degenerate` attribute on a multibit black box, a single-bit cell by that name must exist in the library.
- For multibit black boxes with the `single_bit_degenerate` attribute, all buses or bundles in a cell must have the same width.
- For multibit black boxes with the `single_bit_degenerate` attribute, the unbused pins, buses, and bundles must match the ports of the single-bit cell, and vice versa.

The library description does not include information such as cell height; this must be provided by the library developer.

Defining Multiplexers

A one-hot MUX is a library cell that behaves functionally as a regular MUX logic gate. However, in the case of a one-hot MUX, some inputs are considered dedicated control inputs and others are considered dedicated data inputs. There are as many control inputs as data inputs, and the function of the cell is the logic AND of the i_{th} control input with the i_{th} data input. For example, a 4-to-1 one-hot MUX has the following function:

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

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

Library Requirements

One-hot MUX library cells must meet the following requirements:

- A one-hot MUX cell in the target library should be a single-output cell.
- Its inputs can be divided into two disjoint sets of the same size as follows:

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

where n is greater than 1 and is the size of the set. Actual names of the inputs can vary.

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

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

or

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

- The cell must have a combinational function FO defined on the output with respect to all its inputs. This function FO must logically define, together with the contention condition, a base function F^* that is a sum of n product terms, where the i_{th} term contains all the inputs in C , with C_i high and all others low and exclusively one input in D .

Examples of the defined function are as follows (for $n = 3$):

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

or

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

The function FO can take many forms, if it satisfies the following condition:

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

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

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

or

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

Note:

When FO is restricted by FC, inverting all inputs in D is equivalent to inverting the output. Inverting only a subset of D yields an incompatible function. It is recommended that you use the simple form described earlier, or F*.

The following example shows a cell that is properly specified.

Example

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

Defining Decoupling Capacitor Cells, Filler Cells, and Tap Cells

Decoupling capacitor cells, or *decap cells*, are cells that have a capacitor placed between the power rail and the ground rail to overcome dynamic voltage drop; filler cells are used to connect the gaps between the cells after placement; and tap cells are physical-only cells that have power and ground pins and do not have signal pins. Tap cells are well-tied cells that bias the silicon infrastructure of n-wells or p-wells. Cell-level attributes that identify decoupling capacitor cells, filler cells, and tap cells in libraries are supported.

Syntax

The following syntax shows a cell with the `is_decap_cell`, `is_filler_cell` and `is_tap_cell` attributes, which identify decoupling capacitor cells, filler cells, and tap cells, respectively. However, only one attribute can be set to true in a given cell.

```
cell (cell_name) {  
    ...  
    is_decap_cell : true | false;  
    is_filler_cell : true | false;  
    is_tap_cell : true | false;  
}  
/* End cell group */
```

Cell-Level Attributes

The following attributes can be set at the cell level to identify decoupling capacitor cells, filler cells, and tap cells.

is_decap_cell

The `is_decap_cell` attribute identifies a cell as a decoupling cell. Valid values are true and false.

is_filler_cell

The `is_filler_cell` attribute identifies a cell as a filler cell. Valid values are true and false.

is_tap_cell

The `is_tap_cell` attribute identifies a cell as a tap cell. Tap cells are physical-only cells, which means they have power and ground pins only and not signal pins. Tap cells are well-tied cells that bias the silicon infrastructure of n-wells or p-wells. Valid values for the `is_tap_cell` attribute are true and false.

8

Defining Sequential Cells

This chapter describes the peculiarities of defining flip-flops and latches, building upon the cell description syntax given in [Chapter 7, “Defining Core Cells.”](#) It describes group statements that apply only to sequential cells and also describes a variation of the `function` attribute that makes use of state variables.

To design flip-flops and latches, you must understand the following concepts and tasks:

- [Using Sequential Cell Syntax](#)
- [Describing a Flip-Flop](#)
- [Using the function Attribute](#)
- [Describing a Multibit Flip-Flop](#)
- [Describing a Latch](#)
- [Describing a Multibit Latch](#)
- [Describing Sequential Cells With the Statetable Format](#)
- [Flip-Flop and Latch Examples](#)
- [Cell Description Examples](#)

Using Sequential Cell Syntax

You can describe sequential cells with the following cell definition formats:

- ff or latch format

Cells using the ff or latch format are identified by the `ff` group and `latch` group.

- statetable format

Cells using the statetable format are identified by the `statetable` group. The statetable format supports all the sequential cells supported by the ff or latch format. In addition, the statetable format supports complex sequential cells, such as the following:

- Sequential cells with multiple clock ports, such as a cell with a system clock and a test scan clock
- internal state sequential cells, such as master-slave cells
- Multistate sequential cells, such as counters and shift registers
- Sequential cells with combinational outputs
- Sequential cells with complex clocking and complex asynchronous behavior
- Sequential cells with multiple simultaneous input transitions
- Sequential cells with illegal input conditions

The statetable format contains a complete, expanded set of table rules for which all L and H permutations of table input are explicitly specified.

Some cells cannot be modeled with the statetable format. For example, you cannot use the statetable format to model a cell whose function depends on differential clocks when the inputs change.

The format you use depends on the tool for which you design the library.

- Design synthesis, test synthesis, and fault simulation tools support only the ff or latch format.

Note:

The Design Compiler tool supports the ff or latch format, and ASIC vendors should continue to use ff or latch format for all sequential cells that are already supported. However, the statetable syntax is more intuitive and easier to use than the current sequential modeling format. Therefore, ASIC vendors should add the statetable format to all their sequential cells, to ensure automatic functional verification and compatibility.

- DFT Compiler supports the statetable format design rule checking (DRC) but requires ff or latch format for scan substitution. See [Chapter 9, “Defining Test Cells,”](#) for information about modeling cells for test.

Describing a Flip-Flop

To describe an edge-triggered storage device, include a ff group or a statetable group in a cell definition. This section describes how to define a flip-flop by using the ff or latch format. See [“Describing Sequential Cells With the Statetable Format” on page 8-24](#) for the way to define cells using the statetable group.

Using the ff Group

A ff group describes either a single-stage or a master-slave flip-flop. The ff_bank group represents multibit registers, such as a bank of flip-flops. See [“Describing a Multibit Flip-Flop” on page 8-12](#) for more information about the ff_bank group.

Syntax

```
library (lib_name) {
    cell (cell_name) {
        ...
        ff ( variable1, variable2 ) {

            clocked_on : "Boolean_expression" ;
            next_state : "Boolean_expression" ;

            clear : "Boolean_expression" ;
            preset : "Boolean_expression" ;

            clear_preset_var1 : value ;
            clear_preset_var2 : value ;
            clocked_on_also : "Boolean_expression" ;
            power_down_function : "Boolean_expression" ;
        }
    }
}
```

variable1

The state of the noninverting output of the flip-flop. It is considered the 1-bit storage of the flip-flop.

variable2

The state of the inverting output

You can name *variable1* and *variable2* anything except the name of a pin in the cell being described. Both variables are required, even if one of them is not connected to a primary output pin.

The `clocked_on` and `next_state` attributes are required in the `ff` group; all other attributes are optional.

clocked_on and clocked_on_also Attributes

The `clocked_on` and `clocked_on_also` attributes identify the active edge of the clock signals.

Single-state flip-flops use only the `clocked_on` attribute. When you describe flip-flops that require both a master and a slave clock, use the `clocked_on` attribute for the master clock and the `clocked_on_also` attribute for the slave clock.

Examples

A rising-edge-triggered device is:

```
clocked_on : "CP";
```

A falling-edge-triggered device is:

```
clocked_on : "CP'";
```

next_state Attribute

The `next_state` attribute is a logic equation written in terms of the cell's input pins or the first state variable, *variable1*. For single-stage storage elements, the `next_state` attribute equation determines the value of *variable1* at the next active transition of the `clocked_on` attribute.

For devices such as a master-slave flip-flop, the `next_state` attribute equation determines the value of the master stage's output signals at the next active transition of the `clocked_on` attribute.

Example

```
next_state : "D";
```

nextstate_type Attribute

The `nextstate_type` attribute is a pin group attribute that defines the type of `next_state` attribute used in the `ff` or `ff_bank` group.

Any pin with the `nextstate_type` attribute must be included in the value of the `next_state` attribute. The Library Compiler tool checks the consistency between the pin's `nextstate_type` attribute and the `next_state` attribute.

Note:

Specify a `nextstate_type` attribute to ensure that the sync set (or sync reset) pin and the D pin of sequential cells are not swapped when instantiated.

Example

```
nextstate_type : data;
```

[Example 8-5 on page 8-11](#) and [Example 8-6 on page 8-13](#) show the use of the `nextstate_type` attribute.

clear Attribute

The `clear` attribute gives the active value for the clear input.

The example defines an active-low clear signal.

Example

```
clear : "CD'" ;
```

For more information about the `clear` attribute, see [“Describing a Single-Stage Flip-Flop” on page 8-8](#).

preset Attribute

The `preset` attribute gives the active value for the preset input.

The example defines an active-high preset signal.

Example

```
preset : "PD'" ;
```

For more information about the `preset` attribute, see [“Describing a Single-Stage Flip-Flop” on page 8-8](#).

clear_preset_var1 Attribute

The `clear_preset_var1` attribute gives the value that `variable1` has when `clear` and `preset` are both active at the same time.

Example

```
clear_preset_var1 : L;
```

For more information about the `clear_preset_var1` attribute, including its function and values, see “[Describing a Single-Stage Flip-Flop](#)” on page 8-8.

clear_preset_var2 Attribute

The `clear_preset_var2` attribute gives the value that *variable2* has when clear and preset are both active at the same time.

Example

```
clear_preset_var2 : L ;
```

For more information about the `clear_preset_var2` attribute, including its function and values, see “[Describing a Single-Stage Flip-Flop](#)” on page 8-8.

power_down_function Attribute

The `power_down_function` attribute specifies the Boolean condition when the cell’s output pin is switched off by the power and ground pins (when the cell is in off mode due to the external power pin states). If the `power_down_function` attribute is a 1, X is assumed on the pin, meaning that the pin is assumed to be in an unknown state.

You specify the `power_down_function` attribute for combinational and sequential cells. For simple or complex sequential cells, the `power_down_function` attribute also determines the condition of the cell’s internal state. Knowing the sequential cell’s internal state is necessary for formal verification tools when they perform equivalence checking because the internal state is what is verified.

Syntax

```
library (name) {
    cell (name) {
        ff (variable1,variable2) {
            //...flip-flop description...
            clear : "Boolean expression" ;
            clear_preset_var1 : L | H | N | T | X ;
            clear_preset_var2 : L | H | N | T | X ;
            clocked_on : "Boolean expression" ;
            clocked_on_also : "Boolean expression" ;
            next_state : "Boolean expression" ;
            preset : "Boolean expression" ;
            power_down_function : "Boolean expression" ;
        }
        ...
    }
    ...
}
```

Example

```

library ("low_power_cells") {
    cell ("retention_dff") {
        pg_pin(VDD) {
            voltage_name : VDD;
            pg_type : primary_power;
        }
        pg_pin(VSS) {
            voltage_name : VSS;
            pg_type : primary_ground;
        }
        pin ("D") {
            direction : "input";
        }
        pin ("CP") {
            direction : "input";
        }
        ff(IQ,IQN) {
            next_state : "D" ;
            clocked_on : "CP" ;
            power_down_function : "!VDD + VSS" ;
        }
        pin ("Q") {
            function : " IQ ";
            direction : "output";
            power_down_function : "!VDD + VSS";
        }
    ...
}
...
}

```

ff Group Examples

The following is an example of the `ff` group for a single-stage D flip-flop.

```

ff(IQ, IQN) {
    next_state : "D" ;
    clocked_on : "CP" ;
}

```

The example defines two variables, IQ and IQN. The `next_state` attribute determines the value of IQ after the next active transition of the `clocked_on` attribute. In the example, IQ is assigned the value of the D input.

For some flip-flops, the next state depends on the current state. In this case, the first state variable, `variable1` (IQ in the example), is used in the `next_state` statement; and the second state variable, IQN, is not used.

For the example, the `ff` attribute group for a JK flip-flop is,

```
ff(IQ,IQN) {
    next_state : "(J K IQ') + (J K') + (J' K' IQ)";
    clocked_on : "CP";
}
```

The `next_state` and `clocked_on` attributes define the synchronous behavior of the flip-flop.

Describing a Single-Stage Flip-Flop

A single-stage flip-flop does not use the optional `clocked_on_also` attribute.

[Table 8-1](#) shows the functions of the attributes in the `ff` group for a single-stage flip-flop.

Table 8-1 Function Table for Single-Stage Flip-Flop

active_edge	clear	preset	variable1	variable2
clocked_on	inactive	inactive	next_state	!next_state
--	active	inactive	0	1
--	inactive	active	1	0
--	active	active	clear_preset_var1	clear_preset_var2

The `clear` attribute gives the active value for the clear input. The `preset` attribute gives the active value for the preset input. For example, the following statement defines an active-low clear signal.

```
clear : "CD'" ;
```

The `clear_preset_var1` and `clear_preset_var2` attributes specify the value for `variable1` and `variable2` when `clear` and `preset` are both active at the same time. Valid values are shown in [Table 8-2](#).

Table 8-2 Valid Values for the clear_preset_var1 and clear_preset_var2 Attributes

Variable values	Equivalence
L	0
H	1
N	No change

Table 8-2 Valid Values for the clear_preset_var1 and clear_preset_var2 Attributes (Continued)

Variable values	Equivalence
T	Toggle the current value from 1 to 0, 0 to 1, or X to X
X	Unknown

If you use both `clear` and `preset`, you must use either `clear_preset_var1`, `clear_preset_var2`, or both. Conversely, if you include `clear_preset_var1`, `clear_preset_var2`, or both, you must use both `clear` and `preset`.

The flip-flop cell is activated whenever `clear`, `preset`, `clocked_on`, or `clocked_on_also` change.

[Example 8-1](#) is a `ff` group for a single-stage D flip-flop with a rising edge, negative clear and preset, and the output pins set to 0 when both `clear` and `preset` are active (low).

Example 8-1 Single-Stage D Flip-Flop

```
ff(IQ, IQN) {
    next_state : "D" ;
    clocked_on : "CP" ;
    clear : "CD'" ;
    preset : "PD'" ;
    clear_preset_var1 : L ;
    clear_preset_var2 : L ;
}
```

[Example 8-2](#) is a `ff` group for a single-stage, rising edge-triggered JK flip-flop with scan input, negative clear and preset, and the output pins set to 0 when `clear` and `preset` are both active.

Example 8-2 Single-Stage JK Flip-Flop

```
ff(IQ, IQN) {
    next_state :"(TE*TI)+(TE'*J*K')+(TE'*J'*K'*IQ)+(TE'*J*K*IQ')" ;
    clocked_on : "CP" ;
    clear : "CD'" ;
    preset : "PD'" ;
    clear_preset_var1 : L ;
    clear_preset_var2 : L ;
}
```

[Example 8-3](#) is a `ff` group for a D flip-flop with synchronous negative clear.

Example 8-3 D Flip-Flop With Synchronous Negative Clear

```
ff(IQ, IQN) {
    next_state : "D * CLR" ;
    clocked_on : "CP" ;
```

}

Describing a Master-Slave Flip-Flop

The specification for a master-slave flip-flop is the same as for a single-stage device, except that it includes the `clocked_on_also` attribute. [Table 8-3](#) shows the functions of the attributes in the `ff` group for a master-slave flip-flop.

Table 8-3 Function Tables for Master-Slave Flip-Flop

active_edge	clear	preset	internal1	internal2	variable1	variable2
clocked_on	inactive	inactive	next_state	!next_state		
clocked_on_also	inactive	inactive			internal1	internal2
	active	active	clear_preset_var1	clear_preset_var2	clear_preset_var1	clear_preset_var2
	active	inactive	0	1	0	1
	inactive	active	1	0	1	0

The `internal1` and `internal2` variables represent the output values of the master stage, and `variable1` and `variable2` represent the output values of the slave stage.

The `internal1` and `internal2` variables have the same value as `variable1` and `variable2`, respectively, when the `clear` and `preset` attributes are both active at the same time.

Note:

You do not need to specify the `internal1` and `internal2` variables, which represent internal stages in the flip-flop.

[Example 8-4](#) shows the `ff` group for a master-slave D flip-flop with a rising-edge sampling, falling-edge data transfer, negative clear and preset, and output values set to high when the `clear` and `preset` attributes are both active.

Example 8-4 Master-Slave D Flip-Flop

```
ff(IQ, IQN) {
    next_state : "D" ;
    clocked_on : "CLK" ;
    clocked_on_also : "CLKN'" ;
    clear : "CDN'" ;
    preset : "PDN'" ;
```

```

    clear_preset_var1 : H ;
    clear_preset_var2 : H ;
}

```

Using the function Attribute

Each storage device output pin needs a `function` attribute statement. Only the two state variables, `variable1` and `variable2`, can be used in the `function` attribute statement for sequentially modeled elements.

[Example 8-5](#) shows a complete functional description of a rising-edge-triggered D flip-flop with active-low clear and preset.

Example 8-5 D Flip-Flop Description

```

cell (dff) {
    area : 1 ;
    pin (CLK) {
        direction : input ;
        capacitance : 0 ;
    }
    pin (D) {
        nextstate_type : data;
        direction : input ;
        capacitance : 0 ;
    }
    pin (CLR) {
        direction : input ;
        capacitance : 0 ;
    }
    pin (PRE) {
        direction : input ;
        capacitance : 0 ;
    }
    ff (IQ, IQN) {
        next_state : "D" ;
        clocked_on : "CLK" ;
        clear : "CLR'" ;
        preset : "PRE'" ;
        clear_preset_var1 : L ;
        clear_preset_var2 : L ;
    }
    pin (Q) {
        direction : output ;
        function : "IQ" ;
    }
    pin (QN) {
        direction : output ;
        function : "IQN" ;
    }
} /* end of cell dff */

```

Flip-flops that have the `function` attribute statements can be

- Inferred from a state machine description or a VHDL or Verilog description
- Translated to flip-flops in a different technology library
- Sized during timing optimization
- Changed to flip-flops of a different type, such as D to JK, or D with preset to D with clear
- Converted to scan cells by DFT Compiler with the `insert_dft` command

Describing a Multibit Flip-Flop

The `ff_bank` group describes a cell that is a collection of parallel, single-bit sequential parts. Each part shares control signals with the other parts and performs an identical function. The `ff_bank` group is typically used to represent multibit registers. It can be used in `cell` and `test_cell` groups.

The syntax is similar to that of the `ff` group; see “[Describing a Flip-Flop](#)” on page 8-3.

Syntax

```
library (lib_name)
{
    cell (cell_name) {
        ...
        pin (pin_name) {
            ...
        }
        bundle (bundle_name) {
            ...
        }
        ff_bank ( variable1, variable2, bits ) {
            clocked_on : "Boolean_expression" ;
            next_state : "Boolean_expression" ;
            clear : "Boolean_expression" ;
            preset : "Boolean_expression" ;
            clear_preset_var1 : value ;
            clear_preset_var2 : value ;
            clocked_on_also : "Boolean_expression" ;
        }
    }
}
```

An input that is described in a `pin` group, such as the `clk` input, is fanned out to each flip-flop in the bank. Each primary output must be described in a `bus` or `bundle` group, and its function statement must include either `variable1` or `variable2`.

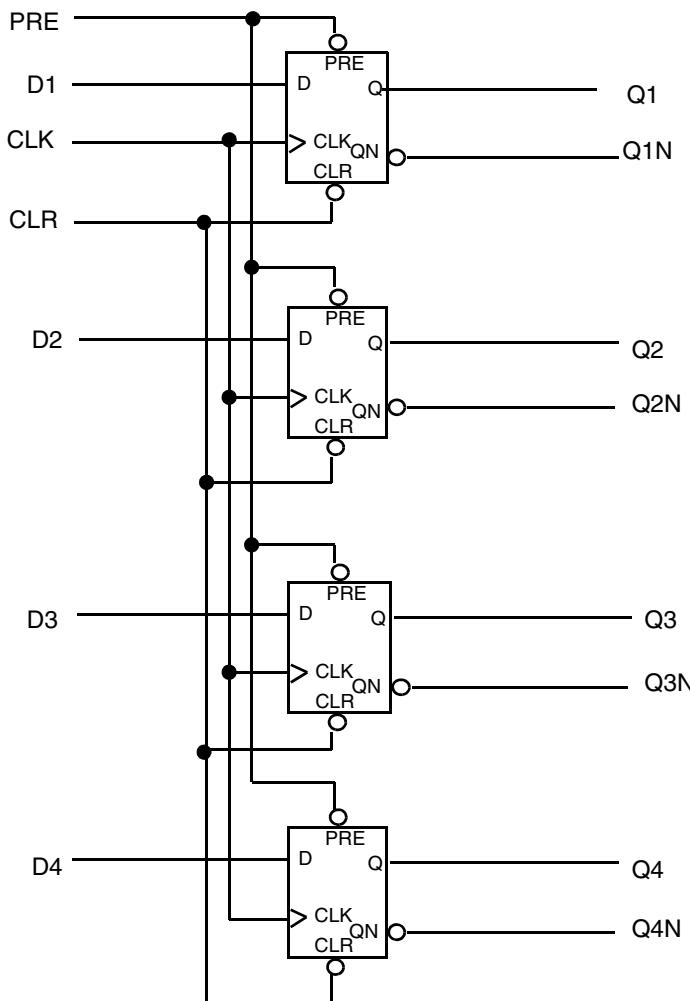
Three-state output pins are allowed; you can add a `three_state` attribute to an output bus or bundle to specify this function.

The `bits` value in the `ff_bank` definition is the number of bits in this multibit cell.

[Figure 8-1](#) shows a multibit register containing four rising-edge-triggered D flip-flops with clear and preset.

[Example 8-6](#) is the description of the multibit register shown in [Figure 8-1](#).

Figure 8-1 Multibit Flip-Flop



Example 8-6 Multibit D Flip-Flop Register

```

cell (dff4) {
    area : 1 ;
    pin (CLK) {

```

```

        direction : input ;
        capacitance : 0 ;
        min_pulse_width_low : 3 ;
        min_pulse_width_high : 3 ;
    }
bundle (D) {
    members(D1, D2, D3, D4);
    nextstate_type : data;
    direction : input ;
    capacitance : 0 ;
    timing() {
        related_pin      : "CLK" ;
        timing_type      : setup_rising ;
    }
    timing() {
        related_pin      : "CLK" ;
        timing_type      : hold_rising ;
    }
}
pin (CLR) {
    direction : input ;
    capacitance : 0 ;
    timing() {
        related_pin      : "CLK" ;
        timing_type      : recovery_rising ;
    }
}
pin (PRE) {
    direction : input ;
    capacitance : 0 ;
    timing() {
        related_pin      : "CLK" ;
        timing_type      : recovery_rising ;
    }
}
ff_bank (IQ, IQN, 4) {
    next_state : "D" ;
    clocked_on : "CLK" ;
    clear : "CLR'" ;
    preset : "PRE'" ;
    clear_preset_var1 : L ;
    clear_preset_var2 : L ;
}
bundle (Q) {
    members(Q1, Q2, Q3, Q4);
    direction : output ;
    function : "(IQ)" ;
    timing() {
        related_pin      : "CLK" ;
        timing_type      : rising_edge ;
    }
    timing() {
        related_pin      : "PRE" ;
    }
}

```

```

        timing_type      : preset ;
        timing_sense    : negative_unate ;
    }
    timing() {
        related_pin    : "CLR" ;
        timing_type    : clear ;
        timing_sense   : positive_unate ;
    }
}
bundle (QN) {
    members(Q1N, Q2N, Q3N, Q4N);
    direction : output ;
    function : "IQN" ;
    timing() {
        related_pin    : "CLK" ;
        timing_type    : rising_edge ;
    }
    timing() {
        related_pin    : "PRE" ;
        timing_type    : clear ;
        timing_sense   : positive_unate ;
    }
    timing() {
        related_pin    : "CLR" ;
        timing_type    : preset ;
        timing_sense   : negative_unate ;
    }
}
} /* end of cell dff4 */

```

Describing a Latch

To describe a level-sensitive storage device, you include a `latch` group or `statetable` group in the cell definition. This section describes how to define a latch by using the `ff` or `latch` format. See “[Describing Sequential Cells With the Statetable Format](#)” on page 8-24 for information about defining cells using the `statetable` group.

latch Group

This section describes a level-sensitive storage device found within a `cell` group.

Syntax

```

library (lib_name) {
    cell (cell_name) {
        ...
        latch (variable1, variable2) {

```

```

        enable : "Boolean_expression" ;
        data_in : "Boolean_expression" ;
        clear : "Boolean_expression" ;
        preset : "Boolean_expression" ;
        clear_preset_var1 : value ;
        clear_preset_var2 : value ;
    }
    pin (pin_name) {
        ...
        data_in_type : data | preset | clear | load ;
    }
}

```

variable1

The state of the noninverting output of the latch. It is considered the 1-bit storage of the latch.

variable2

The state of the inverting output.

You can name *variable1* and *variable2* anything except the name of a pin in the cell being described. Both variables are required, even if one of them is not connected to a primary output pin.

If you include both `clear` and `preset`, you must use either `clear_preset_var1`, `clear_preset_var2`, or both. Conversely, if you include `clear_preset_var1`, `clear_preset_var2`, or both, you must use both `clear` and `preset`.

enable and data_in Attributes

The `enable` and `data_in` attributes are optional, but if you use one of them, you must include the other. The `enable` attribute gives the state of the enable input, and the `data_in` attribute gives the state of the data input.

Example

```
enable : "G" ;
data_in : "D";
```

data_in_type Attribute

In a pin group, the `data_in_type` attribute specifies the type of input data defined by the `data_in` attribute. The valid values are `data`, `preset`, `clear`, or `load`. The default is `data`.

Note:

The Boolean expression of the `data_in` attribute must include the pin with the `data_in_type` attribute.

Example

```
data_in_type : data ;
```

clear Attribute

The `clear` attribute gives the active value for the clear input.

Example

This example defines an active-low clear signal.

```
clear : "CD'" ;
```

preset Attribute

The `preset` attribute gives the active value for the preset input.

Example

This example defines an active-low preset signal.

```
preset : "R'" ;
```

clear_preset_var1 Attribute

The `clear_preset_var1` attribute gives the value that *variable1* has when clear and preset are both active at the same time. Valid values are shown in [Table 8-4](#).

Example

```
clear_preset_var1 : L;
```

Table 8-4 Valid Values for the `clear_preset_var1` and `clear_preset_var2` Attributes

Variable values	Equivalence
L	0
H	1
N	No change
T	Toggle the current value from 1 to 0, 0 to 1, or X to X
X	Unknown

clear_preset_var2 Attribute

The `clear_preset_var2` attribute gives the value that `variable2` has when `clear` and `preset` are both active at the same time. Valid values are shown in [Table 8-4](#).

Example

```
clear_preset_var2 : L ;
```

[Table 8-5](#) shows the functions of the attributes in the `latch` group.

Table 8-5 Function Table for latch Group Attributes

enable	clear	preset	variable1	variable2
active	inactive	inactive	data_in	!data_in
--	active	inactive	0	1
--	inactive	active	1	0
--	active	active	clear_preset_var1	clear_preset_var2

The latch cell is activated whenever `clear`, `preset`, `enable`, or `data_in` changes.

[Example 8-7](#) shows a `latch` group for a D latch with active-high `enable` and negative `clear`.

Example 8-7 D Latch With Active-High enable and Negative clear

```
latch(IQ, IQN) {
    enable : "G" ;
    data_in : "D" ;
    clear : "CD'" ;
}
```

[Example 8-8](#) shows a `latch` group for an SR latch. The `enable` and `data_in` attributes are not required for an SR latch.

Example 8-8 SR Latch

```
latch(IQ, IQN) {
    clear : "S'" ;
    preset : "R'" ;
    clear_preset_var1 : L ;
    clear_preset_var2 : L ;
}
```

Determining a Latch Cell's Internal State

You can use the `power_down_function` attribute to specify the Boolean condition under which the cell's output pin is switched off by the state of the power and ground pins (when the cell is in off mode due to the external power pin states). The attribute should be set under the `latch` group. The `power_down_function` attribute also determines the condition of the cell's internal state.

For more information about `power_down_function`, see “[power_down_function Attribute](#)” on page 8-6.

power_down_function Syntax For Latch Cells

```
library (name) {
    cell (name) {
        latch (variable1,variable2) {
            //...latch description...
            clear : "Boolean expression" ;
            clear_preset_var1 : L | H | N | T | X ;
            clear_preset_var2 : L | H | N | T | X ;
            data_in : "Boolean expression" ;
            enable : "Boolean expression" ;
            preset : "Boolean expression" ;
            power_down_function : "Boolean expression" ;
        }
        ...
    }
}
```

Describing a Multibit Latch

The `latch_bank` group describes a cell that is a collection of parallel, single-bit sequential parts. Each part shares control signals with the other parts and performs an identical function. The `latch_bank` group is typically used in `cell` and `test_cell` groups to represent multibit registers.

latch_bank Group

The syntax is similar to that of the `latch` group. See “[Describing a Latch](#)” on page 8-15.

Syntax

```
library (lib_name) {
    cell (cell_name) {
        ...
        pin (pin_name) {
            ...
        }
    }
}
```

```

        }
    bundle (bus_name) {
        ...
    }
    latch_bank (variable1, variable2, bits) {
        enable : "Boolean_expression" ;
        data_in : "Boolean_expression" ;
        clear : "Boolean_expression" ;
        preset : "Boolean_expression" ;
        clear_preset_var1 : value ;
        clear_preset_var2 : value ;
    }
}

```

An input that is described in a `pin` group, such as the `clk` input, is fanned out to each latch in the bank. Each primary output must be described in a `bus` or `bundle` group, and its function statement must include either `variable1` or `variable2`.

Three-state output pins are allowed; you can add a `three_state` attribute to an output bus or bundle to define this function.

The `bits` value in the `latch_bank` definition is the number of bits in the multibit cell.

Example 8-9 shows a `latch_bank` group for a multibit register containing four rising-edge-triggered D latches.

Example 8-9 Multibit D Latch

```

cell (latch4) {
    area: 16;
    pin (G) { /* gate enable signal, active-high */
        direction : input;
        ...
    }
    bundle (D) { /* data input with four member pins */

        members(D1, D2, D3, D4);
        /*must be first bundle attribute*/
        direction : input;
        ...
    }
    bundle (Q) {
        members(Q1, Q2, Q3, Q4);
        direction : output;
        function : "IQ" ;
        ...
    }
    bundle (QN) {
        members (Q1N, Q2N, Q3N, Q4N);
        direction : output;
        function : "IQN";
        ...
    }
}

```

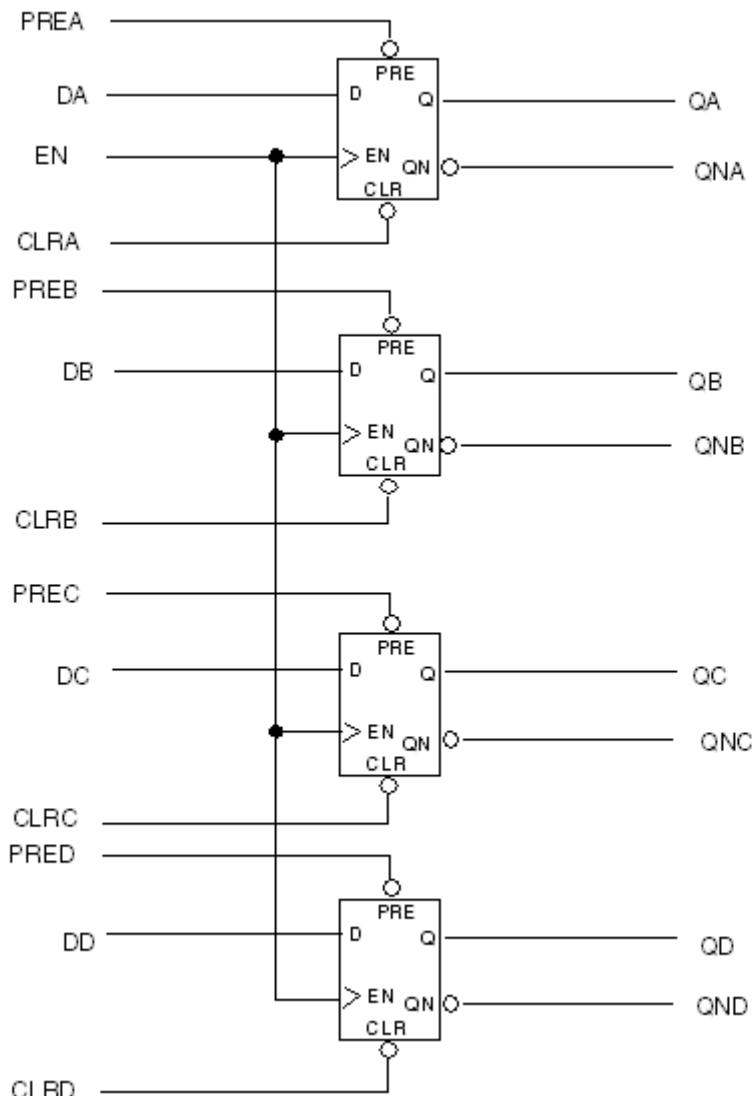
```

        }
        latch_bank(IQ, IQN, 4) {
            enable : "G" ;
            data_in : "D" ;
        }
        ...
    }
}

```

[Figure 8-2](#) shows a multibit register containing four high-enable D latches with clear.

Figure 8-2 Multibit Latch



Example 8-10 is the cell description of the multibit register shown in [Figure 8-2](#) that contains four high-enable D latches with clear.

Example 8-10 Multibit Latches With clear

```
cell (DLT2) {

/* note: 0 hold time */

area : 1 ;
pin (EN) {
    direction : input ;
    capacitance : 0 ;
    min_pulse_width_low : 3 ;
    min_pulse_width_high : 3 ;
}

bundle (D) {
    members(DA, DB, DC, DD);
    direction : input ;
    capacitance : 0 ;
    timing() {
        related_pin      : "EN" ;
        timing_type      : setup_falling ;
    }
    timing() {
        related_pin      : "EN" ;
        timing_type      : hold_falling ;
    }
}

bundle (CLR) {
    members(CLRA, CLRB, CLRC, CLRD);
    direction : input ;
    capacitance : 0 ;
    timing() {
        related_pin      : "EN" ;
        timing_type      : recovery_falling ;
    }
}

bundle (PRE) {
    members(PREA, PREB, PREC, PRED);
    direction : input ;
    capacitance : 0 ;
    timing() {
        related_pin      : "EN" ;
        timing_type      : recovery_falling ;
    }
}

latch_bank(IQ, IQN, 4) {
    data_in : "D" ;
```

```

enable  : "EN" ;
clear   : "CLR ' " ;
preset  : "PRE ' " ;
clear_preset_var1 : H ;
clear_preset_var2 : H ;
}

bundle (Q) {
members(QA, QB, QC, QD);
direction : output ;
function : "IQ" ;
timing() {
    related_pin      : "D" ;
}
timing() {
    related_pin      : "EN" ;
    timing_type      : rising_edge ;
}
timing() {
    related_pin      : "CLR" ;
    timing_type      : clear ;
    timing_sense     : positive_unate ;
}
timing() {
    related_pin      : "PRE" ;
    timing_type      : preset ;
    timing_sense     : negative_unate ;
}
}
bundle (QN) {
members(QNA, QNB, QNC, QND);
direction : output ;
function : "IQN" ;
timing() {
    related_pin      : "D" ;
}
timing() {
    related_pin      : "EN" ;
    timing_type      : rising_edge ;
}
timing() {
    related_pin      : "CLR" ;
    timing_type      : preset ;
    timing_sense     : negative_unate ;
}
timing() {
    related_pin      : "PRE" ;
    timing_type      : clear ;
    timing_sense     : positive_unate ;
}
}
}
} /* end of cell DLT2 */

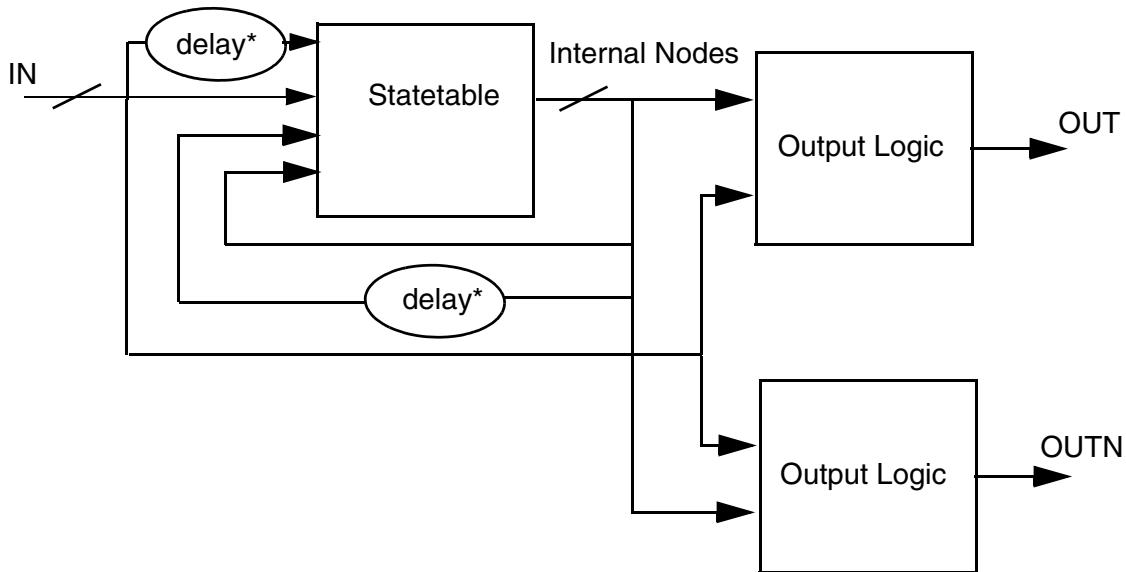
```

Describing Sequential Cells With the Statetable Format

The statetable format provides an intuitive way to describe the function of complex sequential cells. Using this format, the library developer can translate a state table in a databook to a cell description.

[Figure 8-3](#) shows how you can model each sequential output port (OUT and OUTN) in a sequential library cell.

Figure 8-3 Generic Sequential Library Cell Model



OUT and OUTN

Sequential output ports of the sequential cell.

IN

The set of all primary input ports in the sequential cell functionally related to OUT and OUTN.

delay*

A small time delay. An asterisk suffix indicates a time delay.

Statetable

A sequential lookup table. The state table takes a number of inputs and their delayed values and a number of internal nodes and their delayed values to form an index to new internal node values. A sequential library cell can have only one state table.

Internal Nodes

As storage elements, internal nodes store the output values of the state table. There can be any number of internal nodes.

Output Logic

A combinational lookup table. For the sequential cells supported in ff or latch format, there are at most two internal nodes and the output logic must be a buffer, an inverter, a three-state buffer, or a three-state inverter.

To capture the function of complex sequential cells, use the `statetable` group in the `cell` group to define the statetable in [Figure 8-3](#). The `statetable` group syntax maps to the truth tables in databooks.

[Figure 8-4](#) is an example of a table that maps a truth table from an ASIC vendor's databook to the statetable syntax. For table input token values, see “[statetable Group](#)” on page [8-26](#).

Figure 8-4 Mapping Databook Truth Table to Statetable

Databook	Meaning	Statetable input	Statetable output
f	Fall	F	N/A
nc	No event	N/A	N ←
r	Rise	R	N/A
tg	Toggle	N/A	(1) ←
u	Undefined	N/A	X ←
x	Don't Care	-	X
-	Not used	-	X

Annotations on the right side of the table:

- No event from current value
- Toggle flag tg (1)
- Unknown

Annotations at the bottom of the table:

- Rising edge (from low to high)
- Don't care
- Falling edge (from high to low)

To map a databook truth table to a statetable, do the following:

1. When the databook truth table includes the name of an input port, replace that port name with the tokens for low/high (L/H).

2. When the databook truth table includes the name of an output port, use L/H for the current value of the output port and the next value of the output port.
3. When the databook truth table has the toggle flag tg (1), use L/H for the current value of the output port and H/L for the next value of the output port.

In the truth table, an output port preceded with a tilde symbol (~) is inverted. Sometimes you must map f to ~R and r to ~F.

statetable Group

The `statetable` group contains a table consisting of a single string.

Syntax

```
statetable("input node names", "internal node names")
{
    table : "input node values : current internal values \
              : next internal values,\n
    input node values : current internal values : next internal values";
}
```

You need to follow these conventions when using a `statetable` group:

- Give nodes unique names.
- Separate node names with white space.
- Place each rule consisting of a set of input node values, current internal values, and next internal values on a separate line, followed by a comma and the line continuation character (\). To prevent syntax errors, the line continuation character must be followed immediately by the next line character.
- Separate node values and the colon delimiter (:) with white space.
- Insert comments only where a character space is allowed. For example, you cannot insert a comment within an H/L token or after the line continuation character (\).

[Figure 8-5](#) shows an example of a `statetable` group.

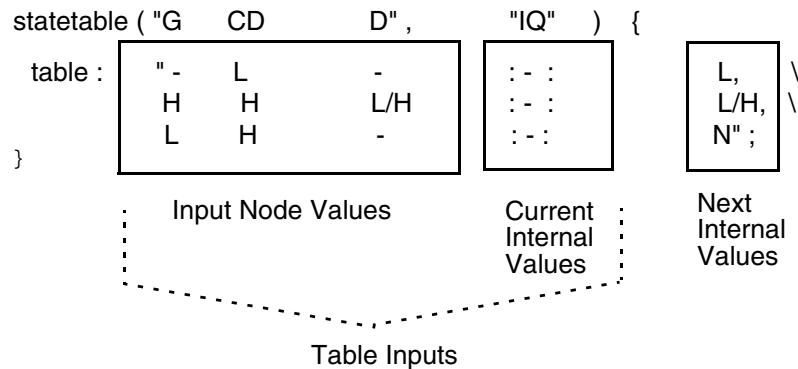
Figure 8-5 statetable Group Example

Table 8-6 shows the token values for table inputs.

Table 8-6 Legitimate Values for Table Inputs (Input and Current Internal Nodes)

Input nodevalue s	Current internal node values	State represented
L	L	Low
H	H	High
-	-	Don't care
L/H	L/H	Expands to both L and H
H/L	H/L	Expands to both H and L
R		Rising edge (from low to high)
F		Falling edge (from high to low)
~R		Not rising edge
~F		Not falling edge

[Table 8-7](#) shows the token values for the next internal node.

Table 8-7 Legitimate Values for Next Internal Node

Next internal node values	State represented
L	Low
H	High
-	Output is not specified
L/H	Expands to both L and H
H/L	Expands to both H and L
X	Unknown
N	No event from current value. Hold. Use only when all asynchronous inputs and clocks are inactive

Note:

It is important to use the N token value appropriately. The output is never N when any input (asynchronous or synchronous) is active. The output should be N only when all the inputs are inactive.

Using Shortcuts

To represent a statetable explicitly, list the next internal node one time for every permutation of L and H for the current inputs, previous inputs, and current states.

[Example 8-11](#) shows a fully expanded `statetable` group for a data latch with active-low clear.

Example 8-11 Fully Expanded statetable Group for Data Latch With Active-Low clear

```

statetable("IQ") {
  table :"
    G      CD      D",
    L      L      L      : L : L, \
    L      L      L      : H : L, \
    L      L      H      : L : L, \
    L      L      H      : H : L, \
    L      H      L      : L : N, \
    L      H      L      : H : N, \
    L      H      H      : L : N, \
    L      H      H      : H : N, \
    H      L      L      : L : L, \
    H      L      L      : H : L, \
    H      L      H      : L : L, \

```

```

H      L      H      : H : L, \
H      H      L      : L : L, \
H      H      L      : H : L, \
H      H      H      : L : H, \
H      H      H      : H : H";
}

```

You can use the following shortcuts when you represent your table.

don't care symbol (-)

For the input and current internal node, the don't care symbol represents all permutations of L and H. For the next internal node, the don't care symbol means the rule does not define a value for the next internal node.

For example, a master-slave flip-flop can be written as follows:

```

statetable("D      CP      CPN", "MQ  SQ") {
    table : "H/L      R      ~F      : - - : H/L      N, \
              - ~R      F      : H/L - : N      H/L, \
              H/LR      F      : L - : H/L      L, \
              H/LR      F      : H - : H/L      H, \
              - ~R      ~F      : - - : N      N";
}

```

Or it can be written more concisely as follows

```

statetable("      D      CP      CPN", "MQ  SQ") {
    table : "      H/L      R      -      : - - - : H/L - , \
              - ~R      -      : - - - : N      - , \
              - -      F      : H/L - - : -      H/L, \
              - -      ~F      : - - - - : -      N";
}

```

L/H and H/L

Both L/H and H/L represent two individual lines: one with L and the other with H. For example, the following line

```
H      H      H/L      : - :      L/H,
```

is a concise version of the following lines.

```
H      H      H      : - :      L,
H      H      L      : - :      H,
```

R, ~R, F, and ~F (input edge-sensitive symbols)

The input edge-sensitive symbols represent permutations of L and H for the delayed input and the current input. Every edge-sensitive input, one that has at least one input edge symbol, is expanded into two level-sensitive inputs: the current input value and the delayed input value. For example, the input edge symbol R expands to an L for the delayed input value and to an H for the current input value. In the following statetable of

a D flip-flop, clock C can be represented by the input pair C* and C. C* is the delayed input value, and C is the current input value.

```
statetable ("C"          "D",    "IQ") {
  table :      "R"   L/H      : - : L/H, \
                ~R   -        : - : N";
```

[Table 8-8](#) shows the internal representation of the same cell.

Table 8-8 Internal Representation

C*	C	D	IQ
L	H	L/H	L/H
H	H	-	N
H	L	-	N
L	L	-	N

Priority ordering of outputs

The outputs follow a prioritized order. Two rules can cover the same input combinations, but the rule defined first has priority over subsequent rules.

Note:

Use shortcuts with care. When in doubt, be explicit.

Partitioning the Cell Into a Model

You can partition a structural netlist to match the general sequential output model described in [Example 8-12](#) by performing these tasks:

1. Represent every storage element by an internal node.
2. Separate the output logic from the internal node. The internal node must be a function of only the sequential cell data input when the sequential cell is triggered.

Internally, the statetable is broken down into specific subtables for each internal node. The amount of memory and processing power required for a subtable is related exponentially to the number of inputs to the subtable. The Library Compiler tool issues a warning when a subtable requires more than 16 inputs; however, a subtable might be of any size if the workstation has the resources to process it.

Note:

There are two ways to specify that an output does not change logic values. One way is to use the inactive N value as the next internal value, and the other way is to have the next internal value remain the same as the current internal value (see the italic lines in [Example 8-12](#)).

Example 8-12 JK Flip-Flop With Active-Low, Direct-Clear, and Negative-Edge Clock

```
statetable ("J K CN CD" , "IQ") {
    table : "
        -   -   -   L   : -   : L, \
        -   -   ~F  H   : -   : N, \
        L   L   F   H   : L/H : L/H, \
        H   L   F   H   : -   : H, \
        L   H   F   H   : -   : L, \
        H   H   F   H   : L/H : H/L" ;
}
```

In [Example 8-12](#), the value of the next internal node of the second rule is N, because both the clear CD and clock CN are inactive. The value of the next internal node of the third rule is L/H, because the clock is active.

Defining an Output pin Group

Every output pin in the cell has either an `internal_node` attribute, which is the name of an internal node, or a `state_function` attribute, which is a Boolean expression of ports and internal pins.

Every output pin can also have an optional `three_state` expression.

An output pin can have one of the following combinations:

- A `state_function` attribute and an optional `three_state` attribute
- An `internal_node` attribute, an optional `input_map` attribute, and an optional `three_state` attribute

state_function Attribute

The `state_function` attribute defines output logic. Ports in the `state_function` Boolean expression must be either input, inout that can be made three-state, or ports with an `internal_node` attribute.

The `state_function` attribute specifies the purely combinational function of input and internal pins. A port in the `state_function` expression refers only to the non-three-state functional behavior of that port. For example, if E is a port of the cell that enables the three-state, the `state_function` attribute cannot have a Boolean expression that uses pin E.

An inout port in the `state_function` expression is treated only as an input port.

Note:

Use the `state_function` attribute to define a cell with a state table. Use this attribute when the functional expression does not reference any state table signals, and is a function of only the input pins.

Example

```
pin(Q)
  direction : output;
  state_function : "A"; /*combinational feedthrough*/
```

internal_node Attribute

The `internal_node` attribute is used to resolve node names to real port names.

The statetable is in an independent, isolated name space. The term *node* refers to the identifiers. Input node and internal node names can contain any characters except white space and comments. These node names are resolved to the real port names by each port with an `internal_node` attribute.

Each output defined by the `internal_node` attribute might have an optional `input_map` attribute.

Example

```
internal_node :
  "IQ";
```

input_map Attribute

The `input_map` attribute maps real port and internal pin names to each table input and internal node specified in the state table. An input map is a listing of port names, separated by white space, that correspond to internal nodes.

Example

```
input_map : "Gx1 CDx1 Dx1 QN"; /*QN is internal node*/
```

Mapping port and internal pin names to table input and internal nodes occurs by using a combination of the `input_map` attribute and the `internal_node` attribute. If a node name is not mapped explicitly to a real port name in the input map, it automatically inherits the node name as the real port name. The internal node name specified by the `internal_node` attribute maps implicitly to the output being defined. For example, to map two input nodes, A and B, to real ports D and CP, and to map one internal node, Z, to port Q, set the `input_map` attribute as follows:

```
input_map : "D  CP  Q"
```

You can use a *don't care* symbol in place of a name to signify that the input is not used for this output. Comments are allowed in the `input_map` attribute.

The delayed nature of outputs specified with the `input_map` attribute is implicit:

Internal node

When an output port is specified for this type of node, the internal node is forced to map to the delayed value of the output port.

Synchronous data input node

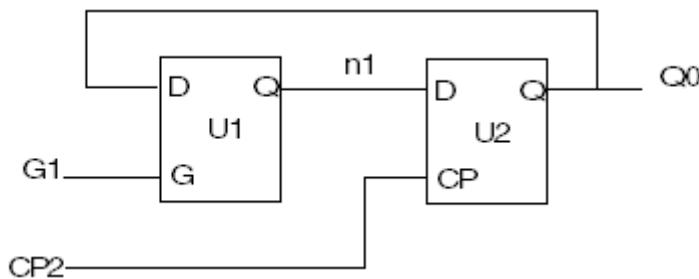
When an output port is specified for this type of node, the input is forced to map to the delayed value of the output port.

Asynchronous or clock input node

When an output port is specified for this type of node, the input is forced to map to the undelayed value of the output port.

For example, [Figure 8-6](#) shows a circuit consisting of latch U1 and flip-flop U2.

Figure 8-6 Circuit With Latch and Flip-Flop



In [Figure 8-6](#), internal pin `n1` should map to ports “`Q0 G1 n1*`” and output `Q0` should map to “`n1* CP2 Q0*`”. The subtle significance is relevant during simultaneous input transitions.

With this `input_map` attribute, the functional syntax for ganged cells is identical to that of nonganged cells. You can use the input map to represent the interconnection of any network of sequential cells (for example, shift registers and counters).

The `input_map` attribute can be completely unspecified, completely specified, or can specify only the beginning ports. The default for an incompletely defined input map is the assumption that missing trailing primary input nodes and internal nodes are equal to the node names specified in the statetable. The current state of the internal node should be equal to the output port name.

inverted_output Attribute

You can define output as inverted or noninverted by using an `inverted_output` attribute on the pin. The `inverted_output` attribute is a required Boolean attribute for the statetable format that you can set for any output port. Set this attribute to false for noninverting output. This is variable1 or IQ for flip-flop or latch groups. Set this attribute to true for inverting output. This is variable2 or IQN for flip-flop or latch groups.

Example

```
inverted_output : true;
```

Some downstream tools make assumptions based on whether an output is considered inverted or noninverted. In the statetable format, an output is considered inverted if it has any data input with a negative sense. This algorithm might be inconsistent with a user's intentions. For example, whether a toggle output is considered inverted or noninverted is arbitrary.

Internal Pin Type

In the flip-flop or latch format, the `pin`, `bus`, or `bundle` groups can have a `direction` attribute with input, output, or inout values. In the statetable format, the `direction` attribute of a pin of a library cell (combinational or sequential) can have the internal value. A pin with the internal value to the `direction` attribute is treated like an output port, except that it is hidden within the library cell. It can take any of the attributes of an output pin, but it cannot have the `three_state` attribute.

An internal pin can have timing arcs and can have timing arcs related to it, allowing a distributed delay model for multistate sequential cells.

Because a cell with a statetable model is considered a black box for synthesis, no checks are performed to ensure consistency between timing and function.

Example

```
pin (n1) {
    direction : internal;
    internal_node :"IQ"; /* use default input_map */
    ...
}
pin (QN) {
    direction : output;
    internal_node :"IQN";
    input_map : "Gx1 CDx1 Dx1 QN"; /* QN is internal node */
    three_state : "EN2";
    ...
}
pin (QNZ) {
```

```

direction : output;
state_function :"QN"; /* ignores QN's three state */

three_state : "EN";
...
}
pin (Y) {
    direction : output;
    state_function : "A"; /* combinational feedthrough */

    ...
}

```

Statetable Checks

In addition to checking for syntax errors, the Library Compiler tool checks for the following errors in `statetable` groups:

- An input that is not functionally required by any of the internal nodes
- An internal node that is purely combinational (that is, the node has no N)
- L/H or H/L present in the output without any L/H or H/L in the inputs
- An input or an internal node name in the statetable that is the same as another input or internal node name in the statetable
- Overlapping entries and unspecified entries, which cause warnings

In `pin` groups, the Library Compiler tool checks for the following errors:

- A functionally related input that is explicitly defined as don't care (-) in the `input_map` attribute
- Too many names defined in the `input_map` attribute
- An internal node name in the `input_map` attribute that does not correspond to the current port name, which results in a warning
- An `internal_node` attribute that does not match any statetable internal node name
- Ports that have missing or illegal attributes
- An input port defined as an internal node in the `input_map` attribute
- Specified internal pins that do not affect any outputs, which results in a warning
- Inclusion of illegal ports in the `state_function` expression; illegal ports are outputs without an `internal_node` attribute and inouts without `three_state` attributes
- Not having any output with an `internal_node` attribute

Timing

With the statetable format, you can specify timing arcs to internal pins.

When you use the ff or latch format in a cell, the Library Compiler tool performs the same timing checks as it does with the statetable format.

When you define only the statetable, the Library Compiler tool checks the timing types only for consistency with other timing types and with the three-state conditions. the Library Compiler tool does not check the timing types against the functional statetable description.

Recommended Statetable Practices

Within the flexibility allowed in specifying the statetable, it is important to follow guidelines consistently for maintainability and legibility. Use the `report_lib` command to verify that the statetable is interpreted correctly.

Beginning users should observe the following recommendations:

- Align the node names and node values for legibility. Align the node values in a column, as shown here:

```
statetable("D    CP"    "IQ") {
    table : "H/L  R    : -   :      H/L,\n           -   ~R    : -   :      N";
}
```

- Use detailed comments when appropriate.

```
statetable("DCP" "IQ") {
    table : "H/L R : - : H/L,/*data capture*/\n           -   ~R : - : N"; /* hold */
}
```

- Place each rule on a separate line.
- Practice with sequential elements that are also specified by the ff or latch format. Verification is automatic.
- Do not use unspecified rules or overlapping rules.
- Do not manually suppress warnings or errors.
- Use real port names in the statetable. Make sure that the node names correspond to the port names. This correspondence reduces the number of names to track.
- Be explicit in rules. Be careful when using shortcuts.
- Be explicit when using `input_map`. If possible, do not use default names.

- Represent complex cells, such as master-slave cells, in one statetable, as shown here:

```
statetable(" D  CP      CPN", "MQSQ") {
    table : " H/L      R      ~F      : -      - : H/L      N, \
              -       ~R      F      : H/L   - : N       H/L, \
              H/L      R      F      : L      - : H/L      L, \
              H/L      R      F      : H      - : H/L      H, \
              -       ~R      ~F      : -      - : N       N";
}
```

- Avoid using internal pins.
- Use the `report_lib` command to double-check the statetable and the table input mapping. For information about using the `report_lib` command, see the *Library Quality Assurance System User Guide*.
- Do not map outputs to edge-sensitive inputs; use a more-complex statetable.
- Do not represent more than one transition per rule.
- Describe test cells only in the ff or latch format.

More experienced users can follow these recommendations:

- Use the don't care (-) shortcuts to reduce the number of rules (see “[Using Shortcuts](#)” on [page 8-28](#)).
- Use rule order priority to reduce the number of rules.

```
statetable("      D      C1      C2", "IQ") {
    table : " H/L      H      R      : -      :      H/L, \
              H/L      R      H      : -      :      H/L, \
              H/L      R      R      : -      :      H/L, \
              -       -       -      : -      :      N";
}
```

- Use the `input_map` attribute to represent networks of tables (for example, shift registers).
- Use the `default input_map` attribute by defining only the minimum number of port names.
- Use the `state_function` attribute for all output logic.
- Define delayed inputs in the statetable and in the `input_map` attribute.
- Filter out warnings.
- Specify internal nodes in the statetable that are not referenced—for example, Q and QN—even though the cell has only a Q output. Such specification makes it easier to reuse statetables among library cells and incurs only a small cost in Library Compiler processing.

Determining a Complex Sequential Cell's Internal State

You can use the `power_down_function` string attribute to specify the Boolean condition under which the cell's output pin is switched off by the state of the power and ground pins (when the cell is in off mode due to the external power pin states). The attribute should be set under the `statetable` group. The `power_down_function` attribute also determines the condition of the cell's internal state.

For more information about the `power_down_function` attribute, see ["power_down_function Attribute" on page 8-6](#).

`power_down_function` Syntax for State Tables

The `power_down_function` attribute is specified after `table`, in the `statetable` group, as shown in the following syntax:

```
statetable( "input node names", "internal node names" ){
    table : " input node values : current internal values : \
              next internal values,\n \
              input node values : current internal values: \
              next internal values" ;
    power_down_function : "Boolean expression" ;
}
```

Flip-Flop and Latch Examples

[Example 8-13](#) through [Example 8-25](#) show flip-flop or latch, and statetable syntax.

Example 8-13 D Flip-Flop

```
/* ff/latch format */

ff (IQ,IQN) {
    next_state : "D" ;
    clocked_on : "CP" ;
}
/* statetable format */

statetable("      D CP", "IQ") {
    table : "      H/L           R   : - :      H/L,\n \
              -             ~R  : - :      N";
}
```

Example 8-14 D Flip-Flops With Master-Slave Clock Input Pins

```
/* ff/latch format */

ff (IQ,IQN) {
    next_state : "D" ;
```

```

    clocked_on : "CK" ;
    clocked_on_also : "CKN'" ;
}

/* statetable format */
statetable(" D   CK      CKN", "MQSQ") {
    table : " H/L   R     ~F  : -  -  : H/L   N, \
              -     ~R     F  : H/L -  : N     H/L, \
              H/L   R     F  : L   -  : H/L   L, \
              H/L   R     F  : H   -  : H/L   H, \
              -     ~R     ~F : -  -  : N     N";
}

```

Example 8-15 D Flip-Flop With Gated Clock

```

/* ff/latch format */

ff (IQ,IQN) {
    next_state : "D" ;
    clocked_on : "C1 * C2" ;
}

/* statetable format */

statetable("      D      C1      C2", "IQ") {
    table : "      H/L      H     R  : -  -  : H/L, \
              H/L      R     H  : -  -  : H/L, \
              H/L      R     R  : -  -  : H/L, \
              -        -     -  : -  -  : N";
}

```

Example 8-16 D Flip-Flop With Active-Low Direct-Clear and Active-Low Direct-Set

```

/* ff/latch format */

ff (IQ,IQN) {
    next_state : "D" ;
    clocked_on : "CP" ;
    clear : " CD' " ;
    preset : " SD' " ;
    clear_preset_var1 : L ;
    clear_preset_var2 : L ;
}

/* statetable format */

statetable("D   CP   CD   SD", "IQ IQN") {
    table : "H/L   R   H   H  : -  -  : H/L   L/H, \
              -     ~R   H   H  : -  -  : N     N, \
              -     -     L   H  : -  -  : L     H, \
              -     -     H   L  : -  -  : H     L, \
              -     -     L   L  : -  -  : L     L";
}

```

```
}
```

Example 8-17 D Flip-Flop With Active-Low Direct-Clear, Active-Low Direct-Set, and One Output

```
/* ff/latch format */

ff (IQ, IQN) {
    next_state : "D" ;
    clocked_on : "CP" ;
    clear : " CD' " ;
    preset : " SD' " ;
    clear_preset_var1 : L ;
    clear_preset_var2 : L ;
}

/* statetable format */

statetable(" D      CP      CD      SD", "IQ") {
    table : " H/L   R      H      H      : - :      H/L,\n           -     ~R     H      H      : - :      N,\n           -     -      L      H/L   : - :      L,\n           -     -      H      L      : - :      H";
}
```

Example 8-18 JK Flip-Flop With Active-Low Direct-Clear and Negative-Edge Clock

```
/* ff/latch format */

ff (IQ, IQN) {
    next_state : " (J' K' IQ) + (J K' ) + (J K IQ' )"
;
    clocked_on : " CN'" ;
    clear : " CD' " ;
}

/* statetable format */

statetable(" J      K      CD      CD", "IQ") {
    table : " -      -      -      L      : - :      L,\n           -      -      ~F     H      : - :      N,\n           L      L      F      H      : L/H :      L/H \\", \n           H      L      F      H      : - :      H,\n           L      H      F      H      : - :      L,\n           H      H      F      H      : L/H :      H/L";
}
```

Example 8-19 D Flip-Flop With Scan Input Pins

```
/* ff/latch format */

ff (IQ, IQN) {
    next_state : " (D TE') + (TI TE)" ;
    clocked_on : "CP" ;
}
```

```
/* statetable format */

statetable(" D      TE    TI    CP",   "IQ") {
    table : " H/L  L  -      R : - : H/L,\n
              -      H   H/L  R : - : H/L,\n
              -      -      - ~R : - : N";
}
```

Example 8-20 D Flip-Flop With Synchronous Clear

```
/* ff/latch format */

ff (IQ, IQN) {
    next_state : "D CR" ;
    clocked_on : "CP" ;
}

/* statetable format */
statetable(" D      CR    CP",   "IQ") {
    table : " H/L  H  R : - : H/L,\n
              -      L  R : - : L,\n
              -      - ~R : - : N";
}
```

Example 8-21 D Latch

```
/* ff/latch format */

latch (IQ, IQN) {
    data_in : "D" ;
    enable : "G" ;
}

/* statetable format */

statetable(" D G",   "IQ") {
    table : " H/L H : - : H/L,\n
              -     L : - : N";
}
```

Example 8-22 SR Latch

```
/* ff/latch format */

latch (IQ, IQN) {
    clear : "R" ;
    preset : "S" ;
    clear_preset_var1 : L ;
    clear_preset_var2 : L ;
}
```

```
/* statetable format */

statetable(" R   S",    "IQ IQN") {
    table : " H   L   : - - : L   H, \
              L   H   : - - : H   L, \
              H   H   : - - : L   L, \
              L   L   : - - : N   N";
}
```

Example 8-23 D Latch With Active-Low Direct-Clear

```
/* ff/latch format */

latch (IQ,IQN) {
    data_in : "D" ;
    enable : "G" ;
    clear : " CD' " ;
}

/* statetable format */

statetable(" D   G   CD",    "IQ") {
    table : " H/L H   H   : - : H/L, \
              -   L   H   : - : N, \
              -   -   L   : - : L";
}
```

Example 8-24 Multibit D Latch With Active-Low Direct-Clear

```
/* ff/latch format */

latch_bank(IQ, IQN, 4) {
    data_in : "D" ;
    enable : "EN" ;
    clear : "CLR'" ;
    preset : "PRE'" ;
    clear_preset_var1 : H ;
    clear_preset_var2 : H ;
}

/* statetable format */

statetable(" D   EN   CL   PRE", "IQ IQN") {
    table : "H/L  H   H   H   : - - - : H/L  L/H, \
              -   L   H   H   : - - - : N   N, \
              -   -   L   H   : - - - : L   H, \
              -   -   H   L   : - - - : H   L, \
              -   -   L   L   : - - - : H   H";
}
```

Example 8-25 D Flip-Flop With Scan Clock

```
statetable(" D   S   CD   SC   CP", "IQ") {
    table : " H/L  -   ~R   R   : - : H/L, \
              -   H/L  R   ~R  : - : H/L, \
              -   -   ~R   ~R  : - : N, \
              -   -   R    R   : - : X";
}
```

Cell Description Examples

Example 8-26 through Example 8-28 illustrate some of the concepts this chapter discusses.

Example 8-26 D Latches With Master-Slave Enable Input Pins

```
cell(ms_latch) {
    area : 16;
    pin(D) {
        direction : input;
        capacitance : 1;
    }
    pin(G1) {
        direction : input;
        capacitance : 2;
    }
    pin(G2) {
        direction : input;
        capacitance : 2;
    }
    pin(mq) {
        internal_node : "Q";
        direction : internal;
        input_map : "D G1";
        timing() {
            related_pin : "G1";
        }
    }
    pin(Q) {
        direction : output;
        function : "IQ";
        internal_node : "Q";
        input_map : "mq G2";
        timing() {
            related_pin : "G2";
        }
    }
    pin(QN) {
        direction : output;
        function : "IQN";
        internal_node : "QN";
        input_map : "mq G2";
        timing() {
```

```

        related_pin : "G2";
    }
}
ff(IQ, IQN) {
    clocked_on : "G1";
    clocked_on_also : "G2";
    next_state : "D";
}
statetable ( "D G", "Q QN" ) {
    table : "L/H H : - - : L/H H/L,\n          - L : - - : N N";
}
}
}

```

Example 8-27 FF Shift Register With Timing Removed

```

cell(shift_reg_ff) {
    area : 16;
    pin(D) {
        direction : input;
        capacitance : 1;
    }
    pin(CP) {
        direction : input;
        capacitance : 2;
    }
    pin (Q0) {
        direction : output;
        internal_node : "Q";
        input_map : "D CP";
    }
    pin (Q1) {
        direction : output;
        internal_node : "Q";
        input_map : "Q0 CP";
    }
    pin (Q2) {
        direction : output;
        internal_node : "Q";
        input_map : "Q1 CP";
    }
    pin (Q3) {
        direction : output;
        internal_node : "Q";
        input_map : "Q2 CP";
    }
    statetable( "D CP", "Q QN" ) {
        table : "- ~R : - - : N N,\n              H/L R : - - : H/L L/H";
    }
}

```

Example 8-28 FF Counter With Timing Removed

```
cell(counter_ff) {
    area : 16;
    pin(reset) {
        direction : input;
        capacitance : 1;
    }
    pin(CP) {
        direction : input;
        capacitance : 2;
    }
    pin (Q0) {
        direction : output;
        internal_node : "Q0";
        input_map : "CP reset Q0 Q1";
    }
    pin (Q1) {
        direction : output;
        internal_node : "Q1";
        input_map : "CP reset Q0 Q1";
    }
    statetable( "CP reset", "Q0 Q1" ) {
        table : "- L : - - : L H,\n          ~R H : - - : N N,\n          R H : L L : H L,\n          R H : H L : L H,\n          R H : L H : H H,\n          R H : H H : L L";
    }
}
```


9

Defining Test Cells

The DFT Compiler tool from Synopsys is a synthesis tool featuring a design-for-test strategy. This tool facilitates the design of highly testable circuits with minimal speed and area overheads, and generates an associated set of test vectors automatically. The DFT Compiler tool can use either a full-scan or a partial-scan methodology to add scan cells to designs and help make designs controllable and observable.

You must add test-specific details of scannable cells to your technology libraries. For example, you must identify scannable flip-flops and latches and select the types of unscannable cells they replace for a given scan methodology. To do this, you must understand the following concepts described in this chapter:

- [Describing a Scan Cell](#)
- [Describing a Multibit Scan Cell](#)
- [Using Sequential Mapping-Based Scan Insertion](#)
- [Scan Cell Modeling Examples](#)

Describing a Scan Cell

Only MUXed flip-flop, clocked scan, and level-sensitive scan device (LSSD) cells that fit one of the supported methodologies can be specified as scan cells in a library. To specify a cell as a scan cell, add the `test_cell` group to the cell description.

Only the nontest mode function of a scan cell is modeled in the `test_cell` group. The nontest operation is described by its `ff`, `ff_bank`, `latch`, or `latch_bank` declaration and `pin` function attributes. This nontest behavior determines which cells can be replaced when the DFT Compiler tool adds scan circuitry to a design in response to the `insert_dft` command (see [Figure 9-1](#)).

Note:

For a description of `pin` groups, see [“Scan Cell With test_cell Group” on page 9-3](#). The `pin` group restrictions are described in [“Pins in the test_cell Group” on page 9-3](#). See [Chapter 8, “Defining Sequential Cells,”](#) for details on `ff`, `ff_bank`, `latch`, and `latch_bank` group statements and their attributes.

The DFT Compiler tool has two modes in which it replaces a cell with its equivalent scan cell:

- Identical function-based scan insertion
- Sequential mapping-based scan insertion

The identical function-based scan insertion method is faster but more restrictive. This method exactly matches the functional description as well as the pins when substituting scan cells.

Note:

Make sure your libraries always support the identical function-based scan procedure.

The sequential mapping-based scan insertion method is more general. It is described in [“Using Sequential Mapping-Based Scan Insertion” on page 9-25](#). Sequential mapping-based scan insertion currently supports only cells that are edge-triggered in nontest mode.

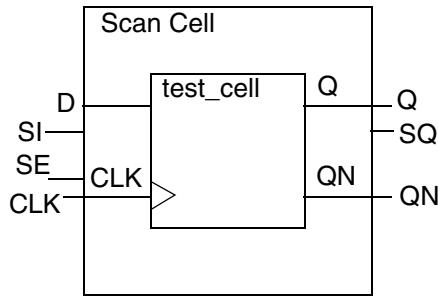
Note:

Set the value of the `read_translate_msff` environment variable to `true` so that the DFT Compiler tool correctly recognizes master-slave latches. If the variable is not set to `true`, the DFT Compiler tool treats these cells as master-slave flip-flop pairs rather than as latch pairs.

test_cell Group

The `test_cell` group defines only the nontest mode function of a scan cell. [Figure 9-1](#) illustrates the relationship between a `test_cell` group and the scan cell in which it is defined.

Figure 9-1 Scan Cell With test_cell Group



There are two important points to remember when defining a scan cell such as the one [Figure 9-1](#) shows:

- Pin names in the scan cell and the test cell must match.
- The scan cell and the test cell must contain the same functional outputs.

Following is the syntax of a `test_cell` group that contains pins:

Syntax

```
library (lib_name) {cell (cell_name) {
    test_cell () {
        ... test cell
description ...
        pin ( name ) {
            ... pin description ...
        }
        pin ( name ) {
            ...
pin description ...
        }
    }
}
```

You do not need to give the `test_cell` group a name, because the test cell takes the cell name of the cell being defined. The `test_cell` group can contain `ff`, `ff_bank`, `latch`, or `latch_bank` group statements and `pin` groups.

Pins in the `test_cell` Group

Both test pins and nontest pins can appear in `pin` groups within a `test_cell` group. These groups are like `pin` groups in a `cell` group but must adhere to the following rules:

- Each pin defined in the `cell` group must have a corresponding pin defined at the `test_cell` group level with the same name.

- The `pin` group can contain only `direction`, `function`, `test_output_only`, and `signal_type` attribute statements. The `pin` group cannot contain timing, capacitance, fanout, or load information.
- The `function` attributes can reflect only the nontest behavior of the cell.
- Input pins must be referenced in an `ff`, `ff_bank`, `latch`, or `latch_bank` statement or have a `signal_type` attribute assigned to them.
- An output pin must have either a `function` attribute, a `signal_type` attribute, or both.

[Example 9-1](#) shows a library model for a multiplexed D flip-flop scan cell with multiple `test_cell` groups.

Example 9-1 Multiplexed D Flip-Flop Scan Cell

```
cell(SDFF1) {
    area : 9;
    pin(D) {
        direction : input;
        capacitance : 1;
        timing() {
            timing_type : setup_rising;
            related_pin : "CLK";
        }
        timing() {
            timing_type : hold_rising;
            related_pin : "CLK";
        }
    }
    pin(CLK) {
        direction : input;
        capacitance : 1;
    }
    pin(SI) {
        direction : input;
        capacitance : 1;
        prefer_tied : "0";
        timing() {
            timing_type : setup_rising;
            related_pin : "CLK";
        }
        timing() {
            timing_type : hold_rising;
            related_pin : "CLK";
        }
    }
    pin(SE) {
        direction : input;
        capacitance : 1;
        timing() {
            timing_type : setup_rising;
            related_pin : "CLK";
        }
    }
}
```

```
        }
        timing() {
            timing_type : hold_rising;
            related_pin : "CLK";
        }
    }
ff("IQ","IQN") {
    next_state : "(D & !SE) | (SI & SE)";
    clocked_on : "CLK";
}
pin(Q) {
    direction : output;
    function : "IQ"
    timing() {
        timing_type : rising_edge;
        related_pin : "CLK";
    }
}
pin(QN) {
    direction : output;
    function : "IQN"
    timing() {
        timing_type : rising_edge;
        related_pin : "CLK";
    }
}
/* first test_cell group defines nontest
   behavior with both Q and QN */
test_cell() {
    pin(D,CLK) {
        direction : input;
    }
    pin(SI) {
        direction : input;
        signal_type : "test_scan_in";
    }
    pin(SE) {
        direction : input;
        signal_type : "test_scan_enable";
    }
    ff ("IQ","IQN") {
        next_state : "D";
        clocked_on : "CLK";
    }
    pin(Q) {
        direction : output;
        function : "IQ";
        signal_type : "test_scan_out";
    }
    pin(QN) {
        direction : output;
        function : "IQN";
        signal_type : "test_scan_out_inverted";
    }
}
```

```

    }
}

/* second test_cell group defines nontest
   behavior with only Q */
test_cell() {
    pin(D,CLK) {
        direction : input;
    }
    pin(SI) {
        direction : input;
        signal_type : "test_scan_in";
    }
    pin(SE) {
        direction : input;
        signal_type : "test_scan_enable";
    }
    ff ("IQ","IQN") {
        next_state : "D";
        clocked_on : "CLK";
    }
    pin(Q) {
        direction : output;
        function : "IQ";
        signal_type : "test_scan_out";
    }
    pin(QN) {
        direction : output;
        /* notice no function attribute for QN pin */
        signal_type : "test_scan_out_inverted";
    }
}

```

test_output_only Attribute

This attribute indicates that an output port is set for test only (as opposed to function only, or function and test).

Syntax

```
test output only : true | false ;
```

When you use statetable format to describe the functionality of a scan cell, you must declare the output pin as set for test only by setting the `test_output_only` attribute to `true`.

Example

```
test output only : true ;
```

signal_type Attribute

Defined in a `test_cell` group, this attribute identifies the type of test pin.

If you use an input pin in both test and nontest modes (such as the clock input in the multiplexed flip-flop methodology), do not include a `signal_type` statement for that pin in the `test_cell` pin definition.

If you use an input pin of a scan cell only in nontest mode and that pin does not exist on the equivalent non scan cell that it replaces, then you must include a `signal_type` statement for that pin in the `test_cell` pin definition.

If you use an output pin in non test mode, it must have a `function` statement. The `signal_type` attribute is used to identify an output pin as a scan-out pin. In a `test_cell` group, the `pin` group for an output pin can contain a `function` statement, a `signal_type` attribute, or both.

Example

```
signal_type : test_scan_in ;
```

For more information about the possible values for the `signal_type` attribute, see the “cell and model Group Description and Syntax” chapter in *Synopsys Logic Library Reference Manual*.

Note:

Do not define a `function` or `signal_type` attribute in the `pin` group if the pin is defined in a previous `test_cell` group of the same cell.

Describing a Multibit Scan Cell

Multibit scan cells can have parallel or serial scan chains. The scan output of a multibit scan cell can reuse the data output pins or have a dedicated scan output (SO) pin in addition to the bus, or the bundle output pins.

Multibit scan cells with the following structures are supported:

- Parallel scan chain without dedicated SO bus
- Parallel scan chain with dedicated SO bus
- Serial scan chain without a dedicated SO pin
- Serial scan chain with a dedicated SO pin

To model multibit scan cells with parallel scan chains and without dedicated SO buses (or scan output combination logic), use the `ff_bank` or `latch_bank` group in the `cell` group.

To model the other multibit scan cell structures, use the `statetable` group in the `cell` group.

Multibit Scan Cell With Parallel Scan Chain

A multibit scan cell with parallel scan chain has parallel data and scan inputs, and parallel functional and scan outputs.

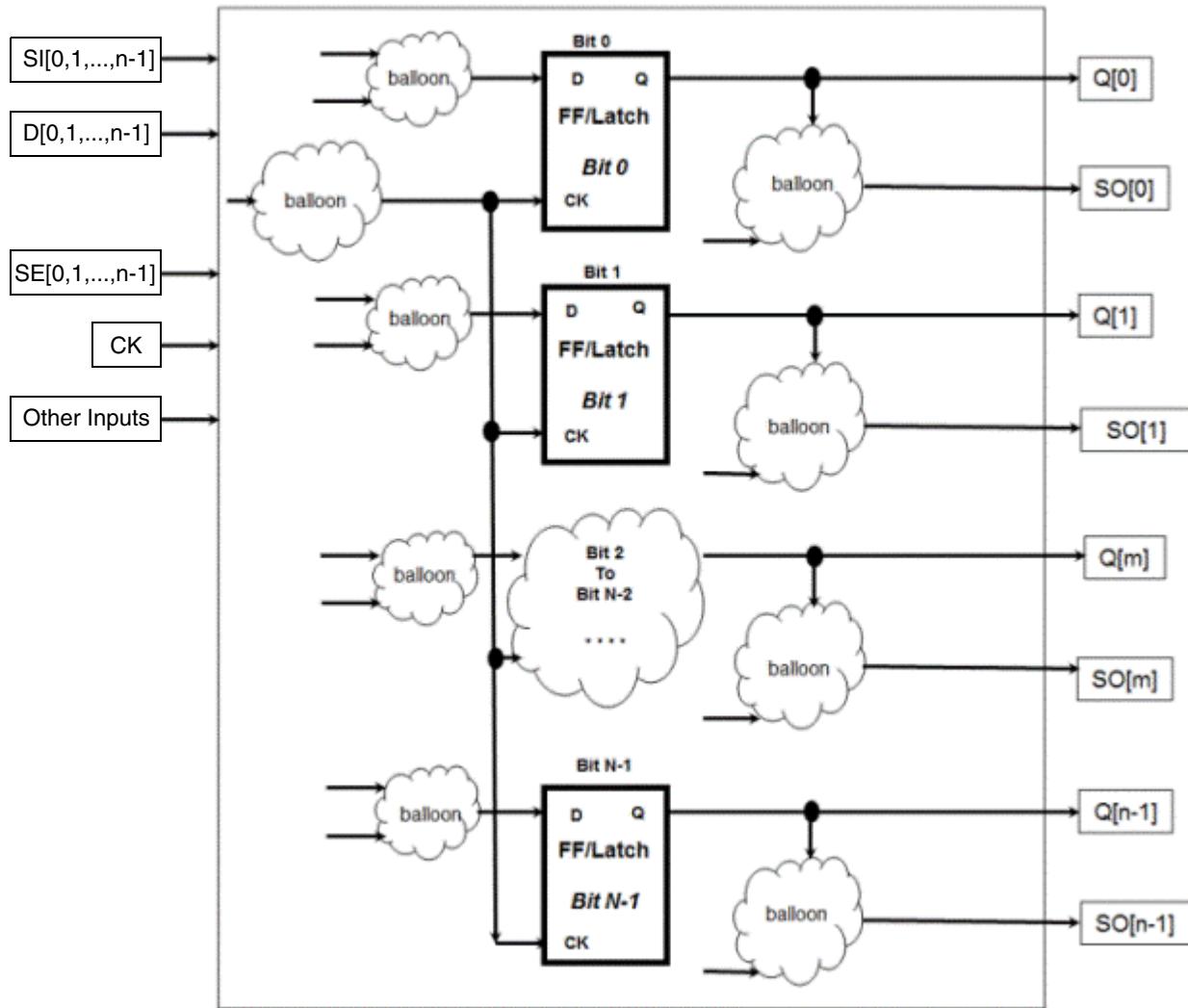
[Figure 9-2](#) shows the schematic diagram of a generic multibit scan cell with parallel input and output buses for the normal and scan modes.

In the normal mode, the cell uses the input and output buses, $D[0:n-1]$ and $Q[0:n-1]$, respectively.

The balloons represent combinational logic. The combinational logic at the input of each sequential element of the cell is a multiplexer with data and scan inputs. At the input of clock, CK, to each sequential element, the combinational logic is a clock pin or a clock-gating logic.

In the scan mode, the cell functions as a parallel-in parallel-out shift register with the scan input bus, $SI[0:n-1]$, and either reuses the bus, $Q[0:n-1]$, or uses a dedicated bus, $SO[0:n-1]$, to output the scan data. The scan enable signal can be a bus, $SE[0:n-1]$, where each bit of the bus enables a corresponding sequential element of the cell. The scan enable signal can also be a single-bit pin to enable each sequential element of the cell.

Figure 9-2 Generic Schematic Diagram of a Multibit Scan Cell With Parallel Scan Chain



Use the following syntax to model the multibit scan cell shown in Figure 9-2:

```
library(library_name) {
...
cell(cell_name) {
...
    bus(scan_in_pin_name) {
        /* cell scan in with signal_type "test_scan_in" from test_cell */
    ...
    }
    bus(scan_out_pin_name) {
        /* cell scan out with signal_type "test_scan_out" from test_cell */
    ...
    }
    bus | bundle (bus_bundle_name) {
...
}
```

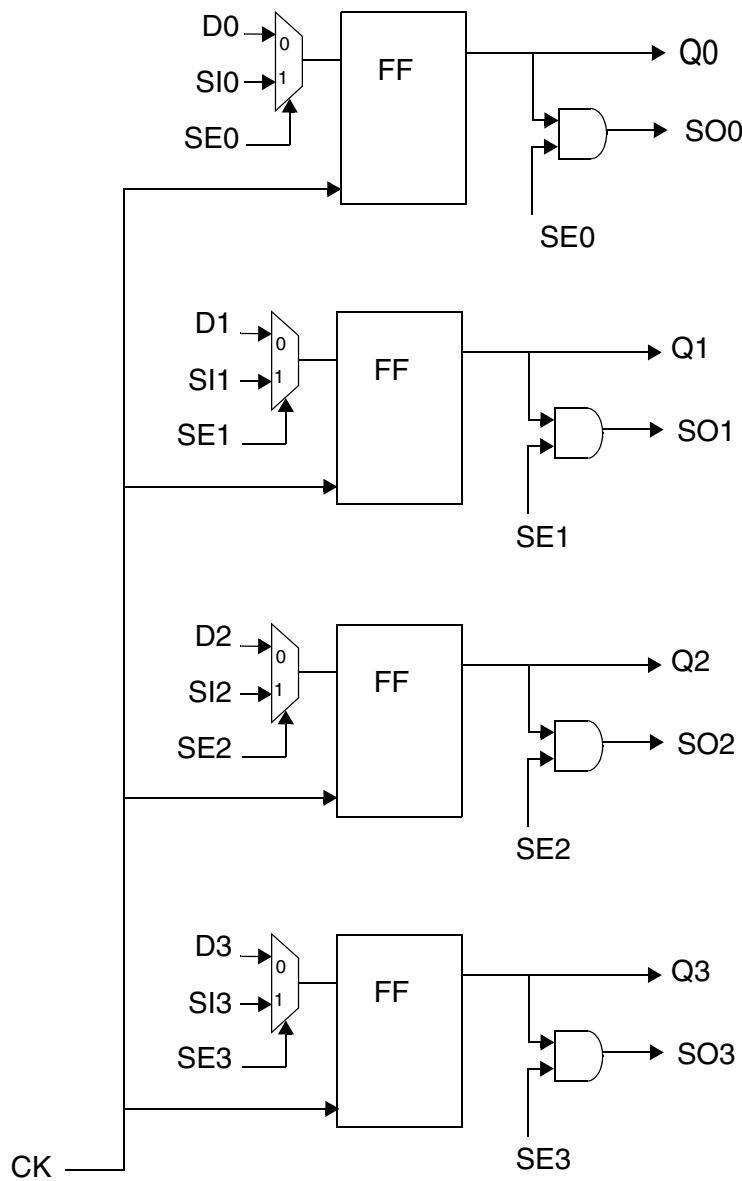
```
        direction : input | output;
    }
    test_cell() {
        pin(scan_in_pin_name) {
            signal_type : test_scan_in;
            ...
        }
        pin(scan_out_pin_name) {
            signal_type : test_scan_out | test_scan_out_inverted;
            ...
        }
        ...
    }
}
```

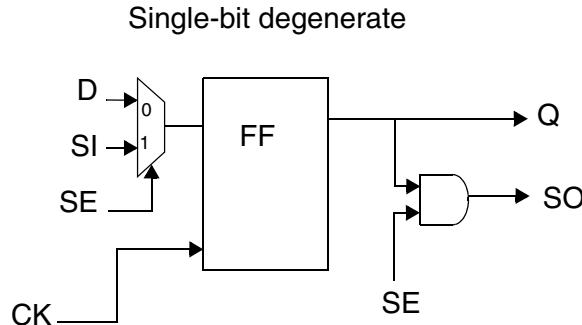
Example

4-bit Scan Cell With Parallel Scan Chains and Dedicated Scan Out Bus

Figure 9-3 shows the schematic of a 4-bit scan cell with parallel scan chains and a dedicated scan output bus, SO[0:3]. The figure also shows an equivalent single-bit cell. During optimization, the Design Compiler tool maps the 4-bit scan cell to four equivalent single-bit cells.

Figure 9-3 4-Bit Scan Cell With Parallel Scan Chain and Single-Bit Equivalent Cell





Each pin of the scan output bus, SO[0:3], has AND logic. The cell is defined using:

- The `statetable` group
- The `state_function` attribute on the bus, SO

In the normal mode, the cell is a parallel shift register that uses the data input bus, D[0:3], and the data output bus, Q[0:3].

In the scan mode, the cell functions as a shift register with parallel scan input, scan enable, and scan output buses, SI[0:3], SE[0:3], and SO[0:3]. To use the cell in the scan mode, set the `signal_type` attribute as `test_scan_out` on the bus, SO, in the `test_cell` group. Do not define the `state_function` attribute for this bus in the `test_cell` group.

[Example 9-2](#) describes a model of the multibit scan cell shown in [Figure 9-3](#). The example specifically models the multibit scan cell, and not the single-bit degenerate cell. The Library Compiler tool identifies the single-bit degenerate cell based on the `state_function` attribute value of the SO bus.

Example 9-2 Model of 4-Bit Scan Cell With Parallel Scan Chain and Dedicated Scan Output

```

cell (4-bit_Parallel_Scan_Cell) {
    ...
    statetable ("D      CK     SI     SE", "IQ, ION") {
        table :   "-      ~R     -      -      :-  -: N     N,  \
                    H/L   R      L      -      :-  -: H/L  H/L, \
                    -      R      H      H/L    :-  -: H/L  L/H";
    }
    /* functional output bus */
    bus(Q) {
        bus_type : bus4;
        direction : output;
        function : IQ;
        ...
    }
    /* dedicated scan output bus */
    bus(SO) {
        bus_type : bus4;
    }
}

```

```
    direction : output;
    state_function : "Q * SE" ;
    ...
}
pin(CK) {
    direction : input;
    ...
}
/* scan enable bus */
bus(SE) {
    bus_type : bus4;
    direction : input;
    ...
}
/* scan input bus */
bus(SI) {
    bus_type : bus4;
    direction : input;
    ...
}
/* data input bus pins */
bus(D) {
    bus_type : bus4;
    direction : input;
    ...
}
...
test_cell () {
    pin(CK){
        direction : input;
    }
    bus(D) {
        bus_type : bus4;
        direction : input;
    }
    bus(SI) {
        bus_type : bus4;
        direction : input;
        signal_type : "test_scan_in";
    }
    bus(SE) {
        bus_type : bus4;
        direction : input;
        signal_type : "test_scan_enable";
    }
    ff_bank (IQ,IQN,4) {
        next_state : "D";
        clocked_on : "CK";
    }
    bus(Q) {
        bus_type : bus4 ;
        direction : output;
        function : "IQ";
```

```
        }
    bus(SO) {
        bus_type : bus4;
        direction : output;
        signal_type : "test_scan_out";
    }
} /* end test_cell group */
}/* end cell group */
```

Multibit Scan Cell With Internal Scan Chain

A multibit scan cell with an internal scan chain has a serial scan chain and a dedicated pin to output the scan signal.

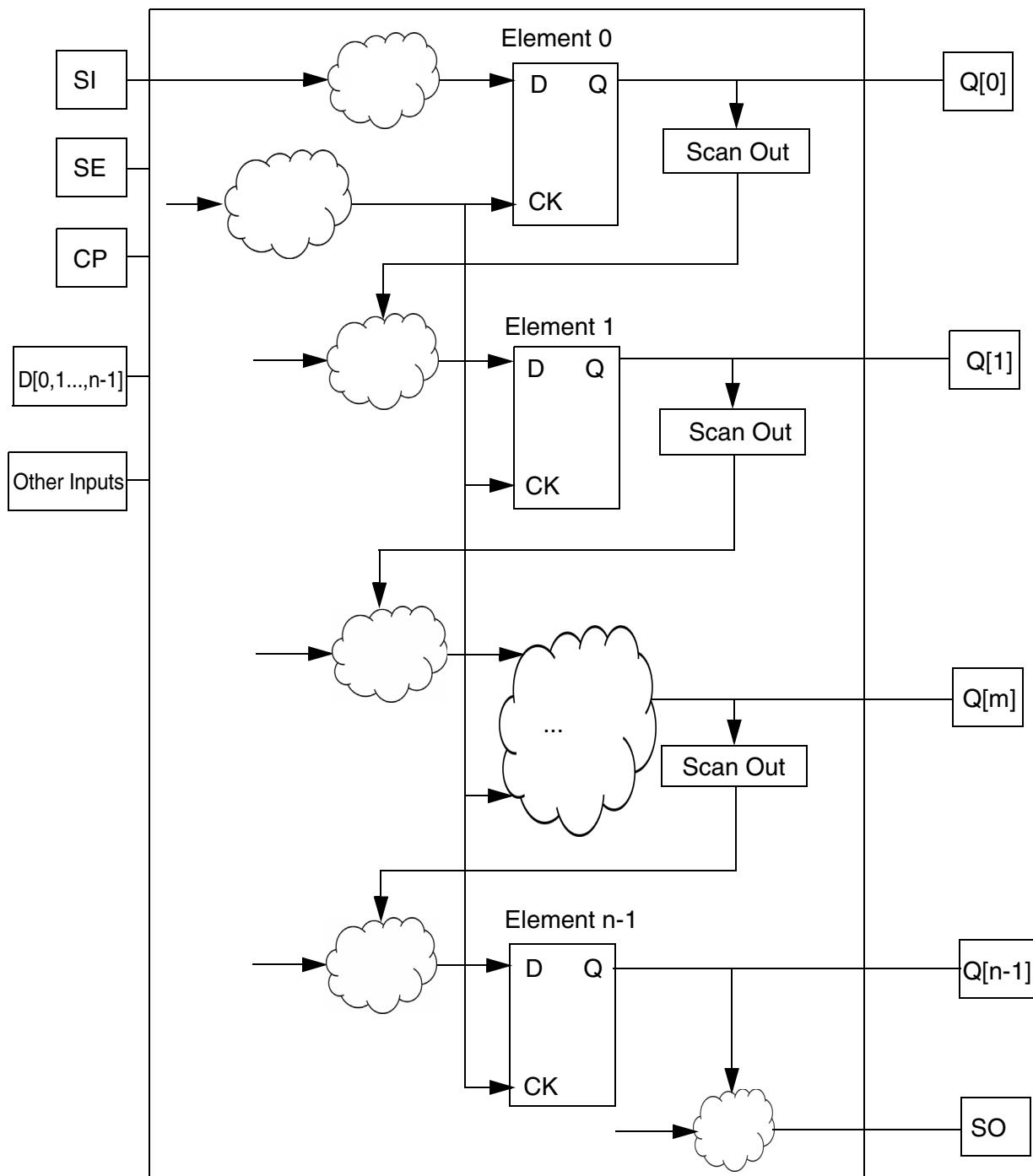
[Figure 9-4](#) shows the schematic diagram of a generic multibit scan cell with an internal or serial scan chain and the pin, SO, for the scan chain output.

In the normal mode, the cell is a shift register that uses the parallel input and output buses, D[0:n-1] and Q[0:n-1], respectively.

The balloons represent combinational logic. The combinational logic at the input of each sequential element of the cell is a multiplexer with data and scan inputs. At the input of clock, CK, to each sequential element, the combinational logic is clock-gating logic.

In the scan mode, the cell functions as a shift register with the single-bit output pin, SO. The serial scan chain is from the scan input (SI) pin to the scan output (SO) pin. The scan chain is stitched using the data output, Q, of each sequential element of the cell.

Figure 9-4 Generic Schematic Diagram of a Multibit Scan Cell With Serial Scan Chain



Use the following syntax to model the multibit scan cell shown in [Figure 9-4](#):

```
library(library_name) {
    ...
    cell(cell_name) {
        pin(scan_in_pin_name) {
            /* cell scan in with signal_type "test_scan_in" from test_cell */
            ...
        }
        pin(scan_out_pin_name) {
            /* cell scan out with signal_type "test_scan_out" from test_cell */
            ...
        }
        bus | bundle (bus_bundle_name) {
            direction : inout | output;
            scan_start_pin : pin_name;          /* optional */
            scan_pin_inverted : true | false; /* optional */
        }
        test_cell() {
            pin(scan_in_pin_name) {
                signal_type : test_scan_in;
                ...
            }
            pin(scan_out_pin_name) {
                signal_type : test_scan_out | test_scan_out_inverted;
                ...
            }
            ...
        }
    }
}
```

Note:

The Library Compiler tool does not support multibit scan cells with internal scan chains where one or more bits of the multibit output bus or bundle have combinational logic.

Attributes Defined in the bus or bundle Group

This section describes the optional `scan_start_pin` and `scan_pin_inverted` attributes specified in the `bus` and `bundle` groups to model the internal scan chain in the multibit scan cell with dedicated scan output pin. For multibit mapping, the Library Compiler tool automatically derives these attributes.

scan_start_pin Attribute

The optional `scan_start_pin` attribute specifies the scan output pin of a sequential element of the multibit scan cell, where the internal scan chain begins.

The tool supports only the following scan chains: from the least significant bit (LSB) to the most significant bit (MSB) of the output `bus` or `bundle` group; and from the MSB to the LSB of the output `bus` or `bundle` group.

Therefore, for a multibit scan cell with internal scan chain, the value of the `scan_start_pin` attribute can either be the LSB, or the MSB output pin.

Specifying the LSB scan output pin as the value of the `scan_start_pin` attribute indicates that the scan signal shifts from the LSB sequential element to the MSB sequential element of the multibit scan cell.

Specifying the MSB scan output pin as the value of the `scan_start_pin` attribute indicates that the scan signal shifts from the MSB sequential element to the LSB sequential element.

Specify the `scan_start_pin` attribute in the `bus` or `bundle` group. You cannot specify this attribute in the `pin` group including the pins of the `bus` and `bundle` group.

scan_pin_inverted Attribute

The optional `scan_pin_inverted` attribute specifies that the scan signal is inverted (after the first sequential element of the multibit scan cell). The valid values are `true` and `false`. The default is `false`.

If you specify the `scan_pin_inverted` attribute value as `true`, you must specify the value of the `signal_type` attribute as `test_scan_out_inverted`.

If you specify the `scan_pin_inverted` attribute, you must specify the `scan_start_pin` attribute in the same `bus` or `bundle` group. You cannot specify this attribute in the `pin` group including the pins in the `bus` and `bundle` group.

For the cell in [Figure 9-4](#), the Library Compiler tool identifies the pins in the internal scan chain based on the values of the `scan_start_pin` and `scan_pin_inverted` attributes and the following conditions. The Design Compiler tool infers the cell using the internal scan pin information.

- In the scan cell, the serial scan data moves only in one of the following two directions:
 - Sequential element 0 → sequential element 1 → ... → sequential element n-1
 - Sequential element n-1 → sequential element n-2 → ... → sequential element 0

Note:

The scan chain cannot be random sequential, such as:

Sequential element 2 → sequential element 0 → sequential element 3 → sequential element 1

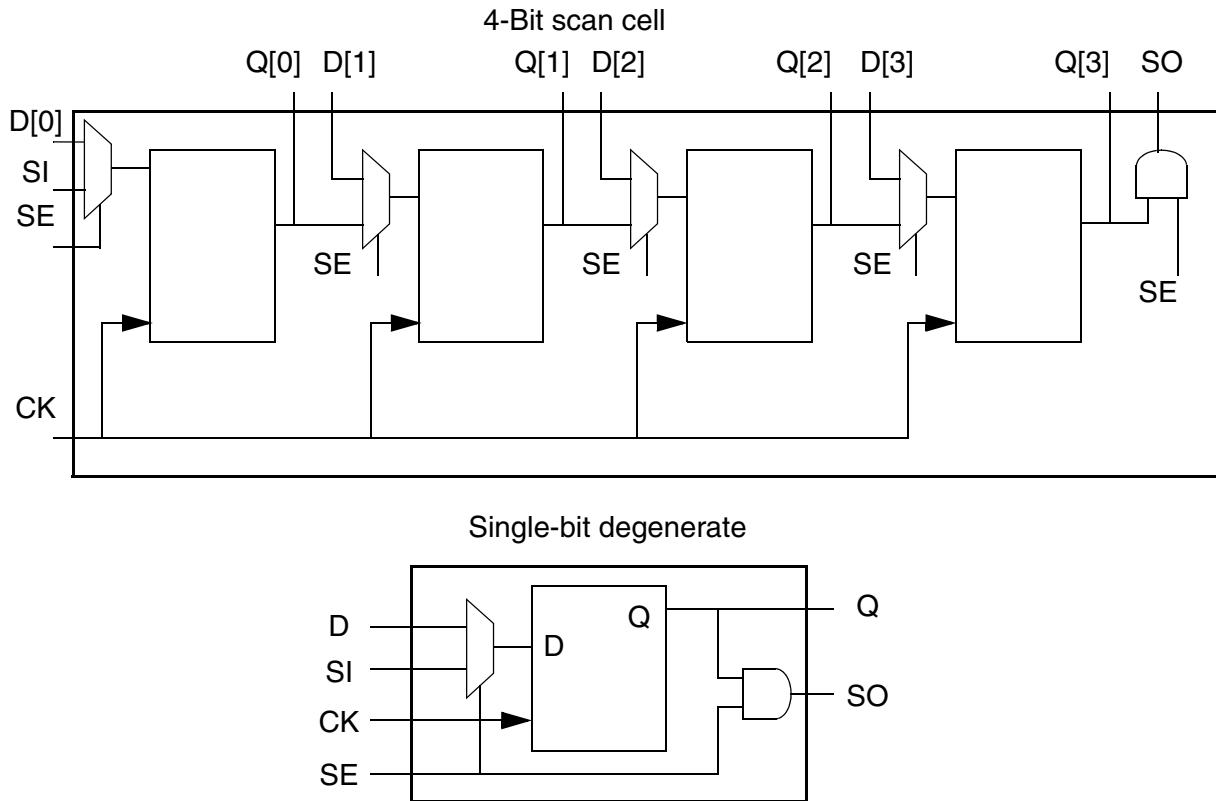
- The scan data shifts between either the inverting output pins or the noninverting output pins of the sequential elements.

Examples

4-Bit Scan Cell With Internal Scan Chain and Dedicated Scan Out Pin

[Figure 9-5](#) shows the schematic of a 4-bit scan cell with a serial scan chain and a scan output pin, SO. The figure also shows a single-bit cell, SB. During optimization, the Design Compiler tool maps the sequential elements of the 4-bit cell to the cell, SB.

Figure 9-5 4-Bit Scan Cell With Internal Scan Chain and Single-Bit Degenerate Cell



The 4-bit cell has the output bus Q[0:3], and a single-bit output pin (SO) with combinational logic. The cell is defined using:

- The `statetable` group
- The `state_function` attribute on the pin, SO

In the normal mode, the cell is a shift register that uses the output bus Q[0:3].

In scan mode, the cell functions as a shift register with the single-bit output pin, SO. To use the cell in scan mode, set the `signal_type` attribute as `test_scan_out` on the pin, SO, in the `test_cell` group. Do not define the `function` attribute of this pin.

Note:

If you define the `function` attribute of the single-bit output pin in a multibit scan cell, the Library Compiler tool issues an error message.

[Example 9-3](#) describes a model of the multibit scan cell shown in [Figure 9-5](#). The example specifically models the multibit scan cell, and not the single-bit degenerate cell. The Library Compiler tool recognizes the single-bit degenerate cell based on the `input_map` and SO bus `state_function` attribute values.

Example 9-3 Model of 4-Bit Multibit Scan Cell With Serial Scan Chain

```

cell (4-bit_Serial_Scan_Chain) {
    ...
    statetable ( " D   CK   SE   SI " ,      "Q" ) {
        table :   " -   ~R   -   -   :   -   : N,   \
                    H/L   R   L   -   :   -   : H/L,   \
                    -   R   H   H/L  :   -   : H/L" ; }

    bus(Q) {
        bus_type : bus4;
        direction : output;
        internal_node: Q;
    /* scan_start_pin : Q[0]; */
        pin (Q[0]) { input_map : " D[0]  CK SE  SI " ; }
        pin (Q[1]) { input_map : " D[1]  CK SE  Q[0]  " ; }
        pin (Q[2]) { input_map : " D[2]  CK SE  Q[1]  " ; }
        pin (Q[3]) { input_map : " D[3]  CK SE  Q[2]  " ; }
        ...
    }
    /* dedicated scan output pin */
    pin(SO) {
        direction : output;
        inverted_output : false;
        state_function : "Q[3] * SE" ;
    ...
    }
    pin(CK) {
        direction : input;
        ...
    }
    pin(SE) {
        direction : input;
        ...
    }
    /* scan input pin */
    pin(SI) {
        direction : input;
        ...
    }
    /* data input bus pins */
    bus(D) {
        direction : input;
        bus_type : bus4;
        ...
    }
    ...
    test_cell () {
        pin(CK){
            direction : input;
        }
        bus(D) {
            bus_type : bus4;
            direction : input;
        }
    }
}

```

```

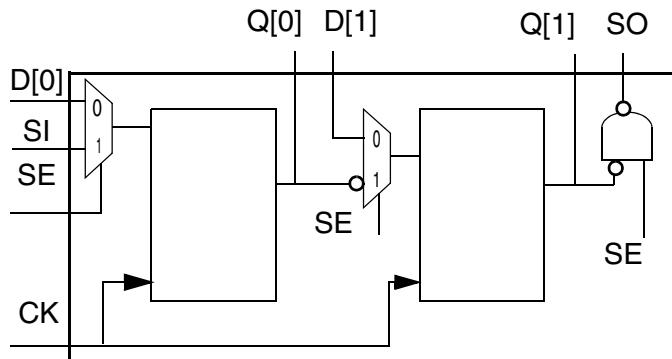
}
pin(SI) {
    direction : input;
    signal_type : "test_scan_in";
}
pin(SE) {
    direction : input;
    signal_type : "test_scan_enable";
}
ff_bank (IQ,IQN,4) {
    next_state : "D";
    clocked_on : "CK";
}
bus(Q) {
    bus_type : bus4 ;
    direction : output;
    function : "IQ";
}
pin(SO) {
    direction : output;
    signal_type : "test_scan_out";
    test_output_only : "true";
}
}
}
}

```

2-Bit Scan Cell With Inverting Scan Out Pin

[Figure 9-6](#) shows the schematic of a 2-bit scan cell with a serial scan chain and an inverting scan output pin, SO.

Figure 9-6 Multibit Scan Cell With an Inverting Scan Out Pin



The following example shows the relevant Liberty syntax to model the cell of [Figure 9-6](#).

```

...
/* scan input pin */
pin(SI){
    ...
}

```

```

/* dedicated scan output pin */
pin(SO) {
    state_function : "!(SE * !Q[0])";
}
/* data input bus pins */
bus(D) {
    ...
}
bus(Q) {
    scan_start_pin : Q[0]; /* optional */
    scan_pin_inverted : true; /* optional */
    pin (Q[0]) {
        internal_node : q;
        input_map : " D[0] CK SE SI ";
    }
    pin (Q[1]) {
        internal_node : qt;
        input_map : " D[1] CK SE Q[0] ";
    }
}
statetable (" D CK SE SI" , "q qt") {
    table : " - ~R - - : - - : N N, \
              H/L R L - : - - : H/L H/L, \
              - R H H/L : - - : H/L L/H" ;
}
...

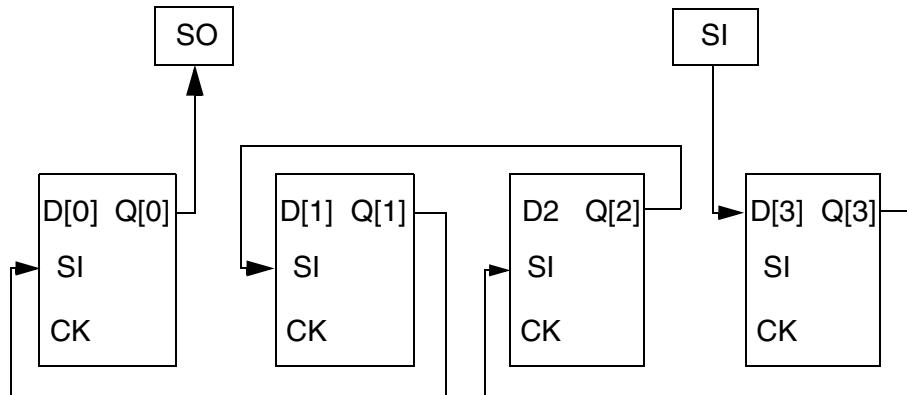
```

The `scan_start_pin` attribute specifies that the internal scan begins at the pin, `Q[0]`. The `scan_pin_inverted` attribute set to `true` specifies that the pins in the scan chain are inverting pins.

4-Bit Scan Cell With Reverse Shifting

[Figure 9-7](#) shows the direction of shift of the scan signal in a 4-bit scan cell.

Figure 9-7 4-bit Scan Cell With Reverse Shifting Scan Signal



The following example shows the relevant `bus` group syntax to model the cell of [Figure 9-7](#):

```
...
bus(Q) {
    bus_type : bus_0_3;
    scan_start_pin : Q[3]; /* optional */
    pin (Q[0]) {
        input_map : " D[0]  CK  SE  Q[1] ";
    }
    pin (Q[1]) {
        input_map : " D[1]  CK  SE  Q[2] ";
    }
    pin (Q[2]) {
        input_map : " D[2]  CK  SE  Q[3] ";
    }
    pin (Q[3]) {
        input_map : " D[3]  CK  SE  SI ";
    }
}
...

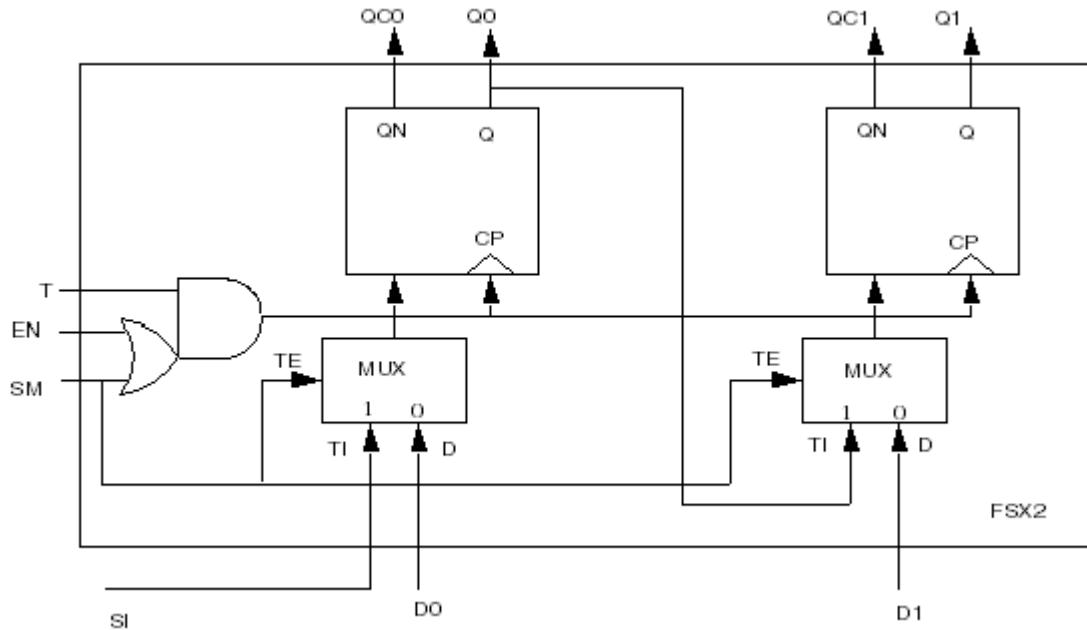
```

The `scan_start_pin` attribute specifies that the internal serial scan begins at the pin, Q[3]. However, the `bus_type` attribute specifies the bus from pin, Q[0], to pin, Q[3]. The scan signal moves from Q[3] to Q[0].

2-Bit Scan Cell Without Dedicated SO Pin

[Figure 9-8](#) shows a 2-bit scan cell with scan and enable inputs, and without a dedicated scan output pin. The output pin, Q1, is reused to output the scan signal.

Figure 9-8 2-Bit Scan Cell With Internal Scan Chain and Without SO Pin



The following example shows the relevant syntax to model the cell of [Figure 9-8](#). In this example, the `statetable` group describes the behavior of a single sequential element of the cell. The `input_map` statements on pin Q0 and pin Q1 indicate the interconnections between the sequential elements—for example, Q of bit 0 goes to TI of bit 1.

```
cell (FSX2) { ...
  bundle (D) {
    members (D0, D1);
    ...
  }
  bundle (Q) {
    scan_start_pin : Q0; /* optional */
    members (Q0, Q1);
    ...
    pin (Q0) {
      input_map : "D0 T SI SM";
      ...
    }
    pin (Q1) {
      input_map : "D1 T Q0 SM";
      ...
    }
  }
  pin (SI) {
    ...
  }
  pin (SM) {
    ...
  }
}
```

```

pin (T) {
    ...
}
pin (EN) {
    ...
}
statetable ( "D CP TI TE EN", "Q QN" ) {
    /*D  CP  TI   TE  EN   Q   QN   Q+   QN+ */
    table : "- ~R - - -:- - -:- N   N,\n"
            "- - - L  L:- - -:- N   N,\n"
            "- R  H/L H -:- - -:- H/L  L/H,\n"
            H/L R   - L  H:- - -:- H/L  L/H "
}
}

```

Using Sequential Mapping-Based Scan Insertion

The Library Compiler tool can insert scan cells based on sequential mapping; see the *DFT Compiler User Guide* for details. Scan insertion uses the sequential mapping algorithms of the Design Compiler tool to find scan equivalent cells. This procedure can be called for any sequential cell that is edge-triggered in normal mode operation.

The sequential mapping-based scan insertion procedure cannot support all scan cells. It can support a cell if at least one `test_cell` in the scan cell has these characteristics:

- Shows nontest mode operation as edge-triggered
- Has a `function` statement for all outputs

Most scan cells that can use sequential mapping-based scan insertion can also use identical function-based scan insertion. Although the identical function-based scan insertion does not work in some cases, it is generally faster than sequential mapping-based scan insertion; therefore, you should use the identical function-based scan insertion where possible.

Sequential mapping-based scan insertion and identical function-based scan insertion have contradictory requirements for scan cells with dedicated scan outputs. Identical function-based scan insertion requires that dedicated scan outputs do not have function statements, even if it is possible to describe them. Such outputs can have only `signal_type` attributes. Sequential mapping-based scan insertion requires that dedicated scan outputs have a `function` statement as well as a `signal_type` attribute. These conflicting requirements can be satisfied by writing two test cells, as shown in the following example.

Example

To support the identical function procedure in the first test cell, use these statements for dedicated scan output:

```

pin (SO) {
    direction : output;
}

```

```

        signal_type : "test_scan_out";
    }

```

To support the sequential mapping procedure in the second test cell, use these statements for dedicated scan output:

```

pin (SO) {
    direction : output;
    signal_type : "test_scan_out";
    function : "IQ";
    test_output_only : true;
}

```

The `function` statement in the second example meets the sequential mapping procedure's requirement that all outputs have a function statement, while the `test_output_only` attribute prevents the output pin from being used as a nontest output.

Scan Cell Modeling Examples

This section contains modeling examples for these test cells:

- Simple multiplexed D flip-flop
- Multibit cells with multiplexed D flip-flop and enable
- LSSD (level-sensitive scan design) scan cell
- Clocked-scan test cell
- Scan D flip-flop with auxiliary clock

Each example contains a complete cell description.

Simple Multiplexed D Flip-Flop

[Example 9-4](#) shows how to model a simple multiplexed D flip-flop test cell.

Example 9-4 Simple Multiplexed D Flip-Flop Scan Cell

```

cell(Sdff1) {
    area : 9;
    pin(D) {
        direction : input;
        capacitance : 1;
        timing() {...}
    }
    pin(CP) {
        direction : input;
        capacitance : 1;

```

```
        timing() {...}
    }
    pin(TI) {
        direction : input;
        capacitance : 1;
        timing() {...}
    }
    pin(TE) {
        direction : input;
        capacitance : 2;
        timing() {...}
    }
    ff(IQ,IQN) {
        /* model full behavior (if possible): */
        next_state : "D TE' + TI TE ";
        clocked_on : "CP";
    }
    pin(Q) {
        direction : output;
        function : "IQ";
        timing() {...}
    }
    pin(QN) {
        direction : output;
        function : "IQN";
        timing() {...}
    }
    test_cell() {
        pin(D) {
            direction : input
        }
        pin(CP) {
            direction : input
        }
        pin(TI) {
            direction : input;
            signal_type : test_scan_in;
        }
        pin(TE) {
            direction : input;
            signal_type : test_scan_enable;
        }
        ff(IQ,IQN) {
            /* just model nontest operation behavior */
            next_state : "D";
            clocked_on : "CP";
        }
        pin(Q) {
            direction : output;
            function : "IQ";
            signal_type : test_scan_out;
        }
        pin(QN) {
```

```
        direction : output;
        function : "IQN";
        signal_type : test_scan_out_inverted;
    }
}
```

Multibit Cells With Multiplexed D Flip-Flop and Enable

[Example 9-5](#) contains a complete library description of a multibit scan cells with multiplexed D flip-flops and enable signal.

Example 9-5 Multibit Scan Cells With Multiplexed D Flip-Flop and Enable

```

library(banktest) {
...
default inout pin_cap : 1.0;
default input pin_cap : 1.0;
default output pin_cap : 0.0;
default fanout load : 1.0;
time unit : "1ns";
voltage unit : "1V";
current unit : "1uA";
pulling resistance unit : "1kohm";
capacitive load unit (0.1,ff);
type (bus4) {
    base type : array;
    data type : bit;
    bit width : 4;
    bit from : 0;
    bit to : 3;
}
cell(FDSX4) {
    area : 36;
    bus(D) {
        bus type : bus4;
        direction : input;
        capacitance : 1;
        timing() {
            timing type : setup_rising;
            related pin : "CP";
        }
        timing() {
            timing type : hold_rising;
            related pin : "CP";
        }
    }
    pin(CP) {
        direction : input;
        capacitance : 1;
    }
}

```

```

pin(TI) {
    direction : input;
    capacitance : 1;
    timing() {
        timing_type : setup_rising;
        related_pin : "CP";
    }
    timing() {
        timing_type : hold_rising;
        related_pin : "CP";
    }
}
pin(TE) {
    direction : input;
    capacitance : 2;
    timing() {
        timing_type : setup_rising;
        related_pin : "CP";
    }
    timing() {
        timing_type : hold_rising;
        related_pin : "CP";
    }
}
statetable ( " D   CP   TI   TE   ", " Q   QN" ) {
    table : " -   ~R   -   -   : -   - : N   N,   \
              -   R   H/L   H   : -   - : H/L   L/H,   \
              H/L   R   -   L   : -   - : H/L   L/H" ;
}
bus(Q) {
    bus_type : bus4;
    direction : output;
    inverted_output : false;
    internal_node : "Q";
    timing() {
        timing_type : rising_edge;
        related_pin : "CP";
    }
    pin(Q[0]) {
        input_map : "D[0] CP TI TE";
    }
    pin(Q[1]) {
        input_map : "D[1] CP Q[0] TE";
    }
    pin(Q[2]) {
        input_map : "D[2] CP Q[1] TE";
    }
    pin(Q[3]) {
        input_map : "D[3] CP Q[2] TE";
    }
}
bus(QN) {
    bus_type : bus4;
}

```

```

direction : output;
inverted_output : true;
internal_node : "QN";
timing() {
    timing_type : rising_edge;
    rise_transition(scalar) {values( " 0.1458 "); }
    fall_transition(scalar) {values( " 0.0523 "); }
    related_pin : "CP";
}
pin(QN[0]) {
    input_map : "D[0] CP TI    TE";
}
pin(QN[1]) {
    input_map : "D[1] CP Q[0] TE";
}
pin(QN[2]) {
    input_map : "D[2] CP Q[1] TE";
}
pin(QN[3]) {
    input_map : "D[3] CP Q[2] TE";
}
}
test_cell() {
    bus (D) {
        bus_type : bus4;
        direction : input;
    }
    pin(CP) {
        direction : input;
    }
    pin(TI) {
        direction : input;
        signal_type : "test_scan_in";
    }
    pin(TE) {
        direction : input;
        signal_type : "test_scan_enable";
    }
    ff_bank("IQ","IQN", 4) {
        next_state : "D";
        clocked_on : "CP";
    }
    bus(Q) {
        bus_type : bus4;
        direction : output;
        function : "IQ";
        signal_type : "test_scan_out";
    }
    bus(QN) {
        bus_type : bus4;
        direction : output;
        function : "IQN";
        signal_type : "test_scan_out_inverted";
    }
}

```

```
        }
    }
}
cell(SCAN2) {
    area : 18;
    bundle(D) {
        members(D0, D1);
        direction : input;
        capacitance : 1;
        timing() {
            timing_type : setup_rising;
            related_pin : "T";
        }
        timing() {
            timing_type : hold_rising;
            related_pin : "T";
        }
    }
    pin(T) {
        direction : input;
        capacitance : 1;
    }
    pin(EN) {
        direction : input;
        capacitance : 2;
        timing() {
            timing_type : setup_rising;
            related_pin : "T";
        }
        timing() {
            timing_type : hold_rising;
            related_pin : "T";
        }
    }
    pin(SI) {
        direction : input;
        capacitance : 1;
        timing() {
            timing_type : setup_rising;
            related_pin : "T";
        }
        timing() {
            timing_type : hold_rising;
            related_pin : "T";
        }
    }
    pin(SM) {
        direction : input;
        capacitance : 2;
        timing() {
            timing_type : setup_rising;
            related_pin : "T";
        }
    }
}
```

```

        timing() {
            timing_type : hold_rising;
            related_pin : "T";
        }
    }
    statetable ( " T      D      EN      SI      SM",           " Q      QN" ) {
        table  : " ~R      -      -      -      : -      -      : N      N , \
                    -      -      L      -      L      : -      -      : N      N , \
                    R      H/L     H      -      L      : -      -      : H/L     L/H, \
                    R      -      -      H/L     H      : -      -      : H/L     L/H";
    }
    bundle(Q) {
        members(Q0, Q1);
        direction : output;
        inverted_output : false;
        internal_node : "Q";
        timing() {
            timing_type : rising_edge;
            related_pin : "T";
        }
        pin(Q0) {
            input_map : "T D0 EN SI SM";
        }
        pin(Q1) {
            input_map : "T D1 EN Q0 SM";
        }
    }
    bundle(QN) {
        members(Q0N, Q1N);
        direction : output;
        inverted_output : true;
        internal_node : "QN";
        timing() {
            timing_type : rising_edge;
            related_pin : "T";
        }
        pin(Q0N) {
            input_map : "T D0 EN SI SM";
        }
        pin(Q1N) {
            input_map : "T D1 EN Q0 SM";
        }
    }
    test_cell() {
        bundle (D) {
            members(D0, D1);
            direction : input;
        }
        pin(T) {
            direction : input;
        }
        pin(EN) {
            direction : input;
        }
    }
}

```

```

    }
    pin(SI) {
        direction : input;
        signal_type : "test_scan_in";
    }
    pin(SM) {
        direction : input;
        signal_type : "test_scan_enable";
    }
    ff_bank("IQ","IQN", 2) {
        next_state : "D";
        clocked_on : "T EN";
    }
    bundle(Q) {
        members(Q0, Q1);
        direction : output;
        function : "IQ";
        signal_type : "test_scan_out";
    }
    bundle(QN) {
        members(Q0N, Q1N);
        direction : output;
        function : "IQN";
        signal_type : "test_scan_out_inverted";
    }
}
}
}

```

LSSD Scan Cell

[Example 9-6](#) shows how to model an LSSD element. For latch-based designs, this form of scan cell has two `test_cell` groups so that it can be used in either single-latch or double-latch mode.

Example 9-6 LSSD Scan Cell

```

cell(LSSD) {
    area : 12;
    pin(D) {
        direction : input;
        capacitance : 1;
        timing() {
            timing_type : setup_falling;
            related_pin : "MCLK";
        }
        timing() {
            timing_type : hold_falling;
            related_pin : "MCLK";
        }
    }
    pin(SI) {

```

```
direction : input;
capacitance : 1;
prefer_tied : "0";
timing() {
    timing_type : setup_falling;
    related_pin : "ACLK";
}
timing() {
    timing_type : hold_falling;
    related_pin : "ACLK";
}
pin(MCLK, ACLK, SCLK) {
    direction : input;
    capacitance : 1;
}
pin(Q1) {
    direction : output;
    internal_node : "Q1";
    timing() {
        timing_type : rising_edge;
        related_pin : "MCLK";
    }
    timing() {
        timing_type : rising_edge;
        related_pin : "ACLK";
    }
    timing() {
        related_pin : "D";
    }
    timing() {
        related_pin : "SI";
    }
}
pin(Q1N) {
    direction : output;
    state_function : "Q1'";
    timing() {
        timing_type : rising_edge;
        related_pin : "MCLK";
    }
    timing() {
        timing_type : rising_edge;
        related_pin : "ACLK";
    }
    timing() {
        related_pin : "D";
    }
    timing() {
        related_pin : "SI";
    }
}
pin(Q2) {
```

```

direction : output;
internal_node : "Q2";
timing() {
    timing_type : rising_edge;
    related_pin : "SCLK";
}
pin(Q2N) {
    direction : output;
    state_function : "Q2'";
    timing() {
        timing_type : rising_edge;
        related_pin : "SCLK";
    }
}
statetable("MCLK D      ACLK SCLK SI",           "Q1      Q2") {
    table :   " L   -   L   -   -   : -   -   : N   - , \
                H   L/H  L   -   -   : -   -   : L/H  - , \
                L   -   H   -   L/H : -   -   : L/H  - , \
                H   -   H   -   -   : -   -   : X   - , \
                -   -   -   L   -   : -   -   : -   N   , \
                -   -   -   H   -   : L/H -   : -   L/H";
}
test_cell() { /* for DLATCH */
    pin(D,MCLK) {
        direction : input;
    }
    pin(SI) {
        direction : input;
        signal_type : "test_scan_in";
    }
    pin(ACLK) {
        direction : input;
        signal_type : "test_scan_clock_a";
    }
    pin(SCLK) {
        direction : input;
        signal_type : "test_scan_clock_b";
    }
    latch ("IQ","IQN") {
        data_in : "D";
        enable : "MCLK";
    }
    pin(Q1) {
        direction : output;
        function : "IQ";
    }
    pin(Q1N) {
        direction : output;
        function : "IQN";
    }
    pin(Q2) {
        direction : output;
    }
}

```

```

        signal_type : "test_scan_out";
    }
    pin(Q2N) {
        direction : output;
        signal_type : "test_scan_out_inverted";
    }
}
test_cell() { /* for MSFF1 */
    pin(D,MCLK,SCLK) {
        direction : input;
    }
    pin(SI) {
        direction : input;
        signal_type : "test_scan_in";
    }
    pin(ACLK) {
        direction : input;
        signal_type : "test_scan_clock_a";
    }
    ff ("IQ","IQN") {
        next_state : "D";
        clocked_on : "MCLK";
        clocked_on_also : "SCLK";
    }
    pin(Q1,Q1N) {
        direction : output;
    }
    pin(Q2) {
        direction : output;
        function : "IQ";
        signal_type : "test_scan_out";
    }
    pin(Q2N) {
        direction : output;
        function : "IQN";
        signal_type : "test_scan_out_inverted";
    }
}
}

```

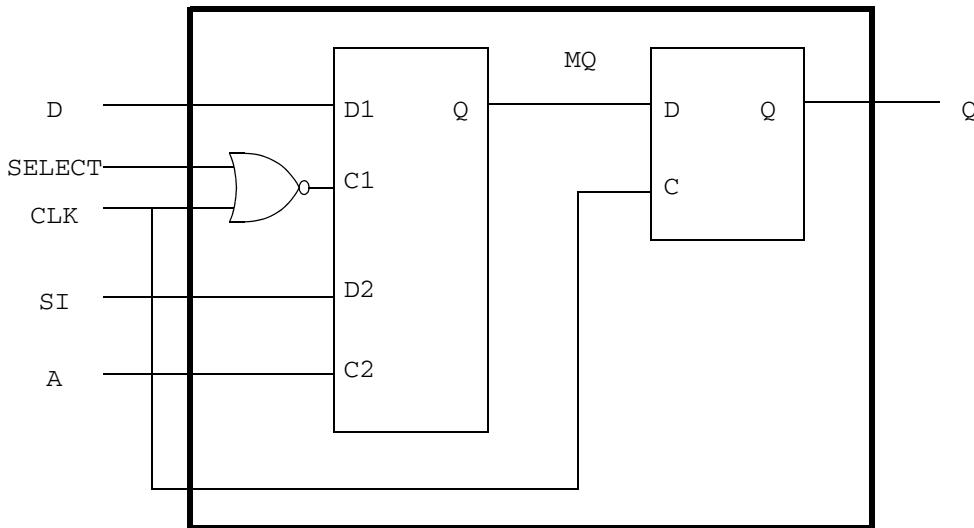
Scan-Enabled LSSD Cell

The scan-enabled LSSD cell is a variation of the LSSD scan cell in the double-latch mode. Unlike the double-latch LSSD scan cell, a single clock controls the enable pins of both the master and slave latches of the scan-enabled LSSD cell.

To recognize the scan-cell type, the `signal_type` attribute is used. If all the values of the `signal_type` attribute, namely, `test_scan_clock_a`, `test_scan_clock_b`, and `test_scan_enable` are present on the pins of a test cell, the corresponding scan cell is

recognized as a scan-enabled LSSD. [Figure 9-9](#) shows the schematic of the scan-enabled LSSD cell.

Figure 9-9 Scan-Enabled LSSD Cell Schematic



Functional Model of the Scan-Enabled LSSD Cell

To model the cell shown in [Figure 9-9](#), define the `signal_type` attribute on the pins of the `test_cell` group, such as the pins, CLK and SELECT. [Example 9-7](#) shows the syntax to define the `signal_type` attribute on the CLK pin. When you set the `signal_type` attribute on the clock pin, CLK, to `test_scan_clock_b`, it indicates that the CLK input also enables the slave-latch in addition to the master-latch of the scan-enabled LSSD cell. In [Figure 9-9](#), the clock pin, CLK, controls both enable pins, C1 and C.

Example 9-7 The signal_type attribute on the Scan-Enabled LSSD Cell CLK pin

```
cell(cell_name) {
    ...
    test_cell() {
        ...
        pin (pin_name) {
            direction : input;
            signal_type : "test_scan_clock_b";
            ...
        }/* End pin group */
    /* End test_cell */
    ...
}
```

[Example 9-8](#) shows the syntax to define the `signal_type` attribute on the select pin, SELECT. When you set the `signal_type` attribute on the select pin, SELECT, to

`test_scan_enable`, the SELECT input is active and enables the scan mode of the LSSD cell. When the SELECT input is inactive, the cell is in the normal mode.

Example 9-8 The `signal_type` Attribute on the Scan-Enabled LSSD Cell SELECT Pin

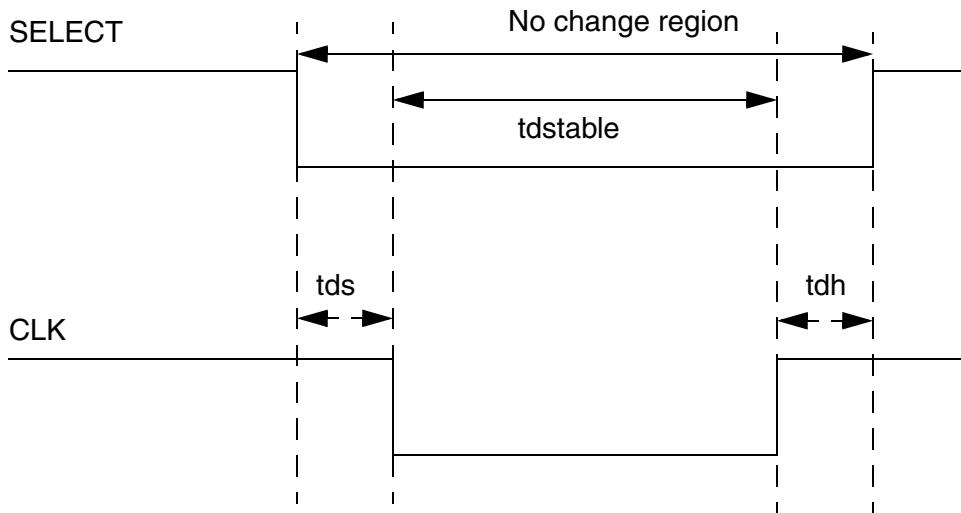
```
cell(cell_name) {
    ...
    test_cell() {
        ...
        pin (pin_name) {
            direction : input;
            signal_type : "test_scan_enable";
        ...
    }/* End pin group */
}/* End test_cell */
...
}
```

Timing Model of the Scan-Enabled LSSD Cell

Figure 9-10 shows a timing arc constraint, from the clock pin, CLK, to the select pin, SELECT, of the scan-enabled LSSD cell. In the normal mode, the constraint ensures that the CLK input works correctly for the duration of the CLK pulse. Therefore, the SELECT input must be stable for the setup period before the CLK pulse (`tds`), when the CLK pulse is active (`tdstable`), and the hold period after (`tdh`) the CLK pulse.

Example 9-9 shows the syntax to model the timing arc, from the clock pin, CLK, to the select pin, SELECT. Timing arcs are modeled by setting the `timing_type` attribute to `nochange` values. As the CLK pin is active-low, set the `timing_type` attribute on the select pin, SELECT, to `nochange_low_low`.

Figure 9-10 A Timing-Arc Constraint of the Scan-Enabled LSSD Cell



Example 9-9 Scan-Enabled LSSD Timing Model Syntax

```
cell(cell_name) {
    ...
    pin (SELECT) {
        direction : input;
        timing() {
            timing_type: "nochange_low_low" ;
            related_pin: "CLK" ;
            fall_constraint(constraint) { /* tds */ }
        ...
        }
        rise_constraint(constraint) { /* tdh */ }
    ...
}
}
...
...
} /* End pin group */
...
}
```

Note:

Do not use the `hold_rising(tds)` and `setup_falling(tdh)` values to model, the clock pin, CLK, to the select pin, SELECT, timing arc. These values of the `timing_type` attribute do not cover the stable region of the SELECT input (`tdstable`).

Scan-Enabled LSSD Cell Model Example

Example 9-10 uses the syntax in Example 9-7, Example 9-8, and Example 9-9 to model the scan-enabled LSSD cell shown in Figure 9-9 on page 9-37.

Example 9-10 Example for the Scan-Enabled LSSD Cell Syntax

```
Cell(scan_enabled_LSSD) {
    ...
    statetable ("CLK D      SELECT   A   SI",           "MQ      Q") {
        table : "
            L     L/H   L       L   -   : -   -   : H/L   -, \
            H     -       -       L   -   : -   -   : N     -, \
            L     -       H       L   -   : -   -   : N     -, \
            L     -       L       H   -   : -   -   : X     -, \
            H     -       L       H   L/H  : -   -   : L/H   -, \
            -     -       H       H   L/H  : -   -   : L/H   -, \
            L     -       -       -   -   : -   -   : -     N, \
            H     -       -       -   -   : L/H  -   : -     L/H";
    }
    pin (CLK) {
        direction : input ;
        timing() {
            timing_type: "min_pulse_width" ;
            related_pin: "CLK" ;
        ...
    }
}
```

```
}

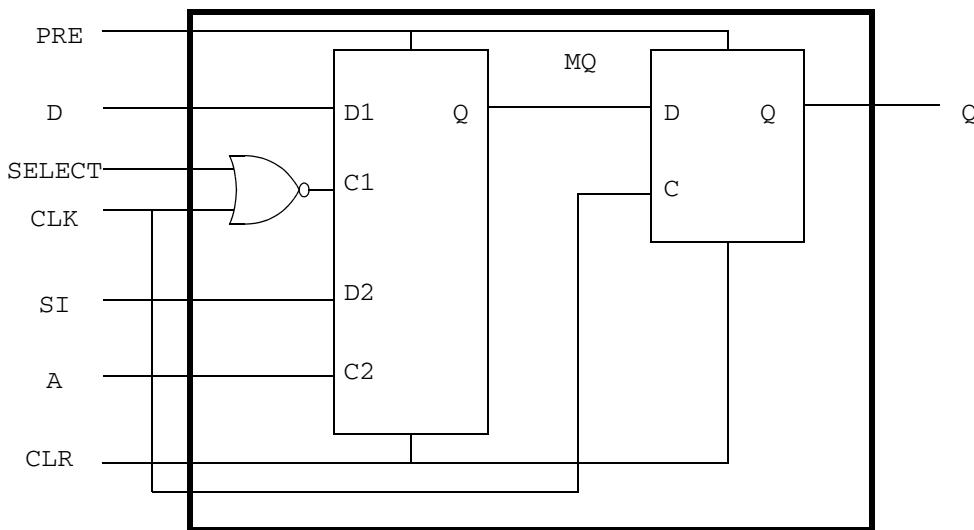
pin (D) {
    direction : input ;
    timing() {
        timing_type: "hold_rising" ;
        related_pin: "CLK" ;
        ...
    }
    timing() {
        timing_type: "setup_rising" ;
        related_pin: "CLK" ;
        ...
    }
}
pin (SELECT) {
    direction : input;
    timing() {
        timing_type: "nochange_low_low" ;
        related_pin: "CLK" ;
        ...
    }
}
pin (SI) {
    direction : input;
    timing() {
        timing_type: "setup_falling" ;
        related_pin: "A" ;
        ...
    }
    timing() {
        timing_type: "hold_falling" ;
        related_pin: "A"
        ...
    }
}
pin (MQ) {
    direction : "internal";
    internal_node : "MQ";
}
pin (Q) {
    direction : output ;
    internal_node: "Q" ;
    timing() {
        timing_type: "rising_edge" ;
        related_pin: "CLK" ;
        ...
    }
}
...
test_cell () {
pin (CLK) {
    direction : input ;
    signal_type : "test_scan_clock_b";
```

```
        }
        pin (D) {
            direction : input ;
        }
        pin (A) {
            direction : input;
            signal_type : "test_scan_clock_a";
        }
        pin ("SELECT") {
            direction : input ;
            signal_type : "test_scan_enable";
        }
        pin (SI) {
            direction : input ;
            signal_type : "test_scan_in";
        }
        pin (Q) {
            direction : output ;
            function : "IQ";
            signal_type : "test_scan_out";
        }
        ff (IQ,IQN) {
            next_state : "D" ;
            clocked_on : "CLK" ;
        }
    }
    ...
}
```

Scan-Enabled LSSD Cell With Asynchronous Inputs

[Figure 9-11](#) shows the schematic of a scan-enabled LSSD cell with asynchronous input preset, PRE, and clear, CLR pins. [Example 9-11](#) shows the modeling syntax for the scan-enabled LSSD cell with the preset, PRE and clear, CLR, input pins.

Figure 9-11 Schematic of a Scan-Enabled LSSD Cell With Asynchronous Inputs



Example 9-11 Modeling Syntax for the Scan-Enabled LSSD Cell With Preset and Clear Inputs

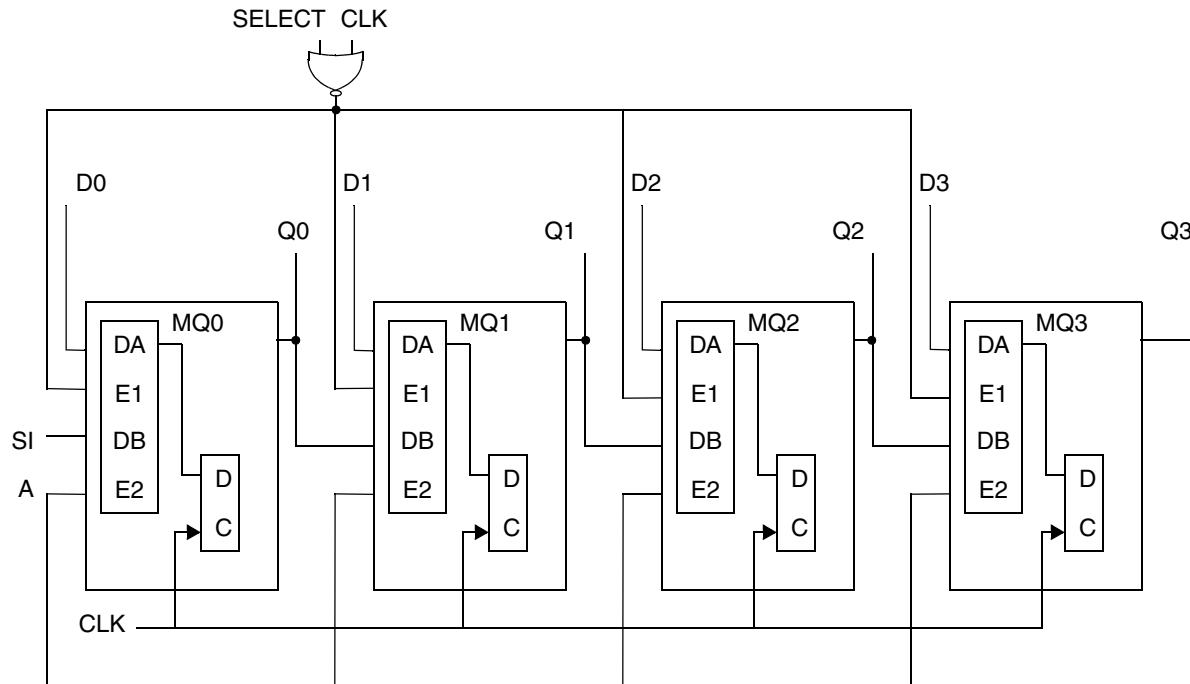
```
cell(LSSD_with_clear_preset) {
...
statetable (" CLK  D   SELECT  A   SI   CLR  PRE",      "MQ   Q") {
table : "
    L   L/H  L     L   -   L   L   : -   - : H/L  -, \
    H   -     -     L   -   L   L   : -   - : N   -, \
    L   -     H     L   -   L   L   : -   - : N   -, \
    L   -     L     H   -   L   L   : -   - : X   -, \
    H   -     L     H   L/H  L   L   : -   - : L/H  -, \
    -   -     H     H   L/H  L   L   : -   - : L/H  -, \
    L   -     -     -   -   L   L   : -   - : -   N, \
    H   -     -     -   -   L   L   : L/H  - : -   L/H, \
    -   -     -     -   -   H   L   : -   - : L   L, \
    -   -     -     -   -   L   H   : -   - : H   H, \
    -   -     -     -   -   H   H   : -   - : L   L";
}
...
test_cell () {
    pin (CLK) {
        direction : input ;
        signal_type : "test_scan_clock_b";
    }
    pin (D) {
        direction : input ;
    }
}
```

```
        }
        pin (CLR) {
            direction : input ;
        }
        pin (PRE) {
            direction : input ;
        }
        pin (A) {
            direction : input;
            signal_type : "test_scan_clock_a";
        }
        pin ("SELECT") {
            direction : input ;
            signal_type : "test_scan_enable";
        }
        pin (SI) {
            direction : input ;
            signal_type : "test_scan_in";
        }
        pin (Q) {
            direction : output ;
            function : "IQ";
            signal_type : "test_scan_out";
        }
        ff (IQ,IQN) {
            next_state : "D" ;
            clocked_on : "CLK" ;
            clear : "CLR" ;
            preset : "PRE" ;
            clear_preset_var1 : L ;
            clear_preset_var2 : L ;
        }
    }
...
}
```

Multibit Scan-Enabled LSSD Cell

[Figure 9-12](#) shows the schematic of a 4-bit scan-enabled LSSD cell. [Example 9-12](#) shows the modeling syntax for the 4-bit scan-enabled LSSD cell.

Figure 9-12 Schematic of a 4-Bit Scan-Enabled LSSD Cell



Example 9-12 4-Bit Scan-Enabled LSSD Cell Modeling Syntax

```
cell(LSSD_multibit) {
...
statetable (" CLK   D   SELECT  A      SI",      "MQ  Q") {
  table : "
    L   L/H L       L   -   : -   -   : H/L -, \
    H   -   -       L   -   : -   -   : N   -, \
    L   -   H       L   -   : -   -   : N   -, \
    L   -   L       H   -   : -   -   : X   -, \
    H   -   L       H   L/H : -   -   : L/H -, \
    -   -   H       H   L/H : -   -   : L/H -, \
    L   -   -       -   -   : -   -   : -   N, \
    H   -   -       -   -   : L/H - : -   L/H";
}
bus(MQ) {
  direction : internal;
  internal_node: MQ;
  bus_type : bus4;
  pin_(MQ[0]) { input_map : "CLK D[0] SELECT A SI"; }
}
```

```

pin (MQ[1]) { input_map : "CLK D[1] SELECT A Q[0]"; }
pin (MQ[2]) { input_map : "CLK D[2] SELECT A Q[1]"; }
pin (MQ[3]) { input_map : "CLK D[3] SELECT A Q[2]"; }
...
}
bus(Q) {
    direction : output;
    internal_node: Q;
    bus_type : bus4;
    pin (Q[0]) { input_map : "CLK D[0] SELECT A SI MQ[0]"; }
    pin (Q[1]) { input_map : "CLK D[1] SELECT A Q[0] MQ[1]"; }
    pin (Q[2]) { input_map : "CLK D[2] SELECT A Q[1] MQ[2]"; }
    pin (Q[3]) { input_map : "CLK D[3] SELECT A Q[2] MQ[3]"; }
...
}
...
test_cell() {
    bus(D) {
        bus_type : bus4;
        direction : input; }
    pin(CLK) {
        direction : input;
        signal_type : test_scan_clock_b; }
    pin(SI) {
        direction : input;
        signal_type : test_scan_in; }
    pin(A) {
        direction : input;
        signal_type : test_scan_clock_a; }
    pin(SELECT) {
        direction : input;
        signal_type : test_scan_enable; }
    ff_bank(IQ,IQN,4) {
        next_state : "D";
        clocked_on : "CLK";
    }
    bus(Q) {
        direction : output;
        bus_type : bus4;
        function : "IQ"; }
    pin(SO) {
        direction : output;
        signal_type : "test_scan_out"; }
} /* End of test_cell */
...
}

```

Clocked-Scan Test Cell

[Example 9-13](#) shows the model of a level-sensitive latch with separate scan clocking. This example shows the scan cell used in clocked-scan implementation.

Example 9-13 Clocked-Scan Test Cell

```
cell(SC_DLATCH) {
    area : 12;
    pin(D) {
        direction : input;
        capacitance : 1;
        timing() {
            timing_type : setup_falling;
            related_pin : "G";
        }
        timing() {
            timing_type : hold_falling;
            related_pin : "G";
        }
    }
    pin(SI) {
        direction : input;
        capacitance : 1;
        prefer_tied : "0";
        timing() {
            timing_type : setup_rising;
            related_pin : "ScanClock";
        }
        timing() {
            timing_type : hold_rising;
            related_pin : "ScanClock";
        }
    }
    pin(G,ScanClock) {
        direction : input;
        capacitance : 1;
    }
    statetable( "D      SI      G ScanClock",      " Q      QN" ) {
        table :   "L/H  -      H      L  : -  - : L/H  H/L,\ \
                    -      L/H  L      R  : -  - : L/H  H/L,\ \
                    -      -      L      ~R : -  - : N      N";
    }
    pin(Q) {
        direction : output;
        internal_node : "Q";
        timing() {
            timing_type : rising_edge;
            related_pin : "G";
        }
        timing() {
            related_pin : "D";
        }
    }
}
```

```
        }
        timing() {
            timing_type : rising_edge;
            related_pin : "ScanClock";
        }
    }
    pin(QN) {
        direction : output;
        internal_node : "QN";
        timing() {
            timing_type : rising_edge;
            related_pin : "G";
        }
        timing() {
            related_pin : "D";
        }
        timing() {
            timing_type : rising_edge;
            related_pin : "ScanClock";
        }
        timing() {
            timing_type : rising_edge;
            related_pin : "ScanClock";
        }
    }
    test_cell() {
        pin(D,G) {
            direction : input;
        }
        pin(SI) {
            direction : input;
            signal_type : "test_scan_in";
        }
        pin(ScanClock) {
            direction : input;
            signal_type : "test_scan_clock";
        }
        pin(SE) {
            direction : input;
            signal_type : "test_scan_enable";
        }
        latch ("IQ","IQN") {
            data_in : "D";
            enable : "G";
        }
        pin(Q) {
            direction : output;
            function : "IQ";
            signal_type : "test_scan_out";
        }
        pin(QN) {
            direction : output;
            function : "IQN";
            signal_type : "test_scan_out_inverted";
        }
    }
}
```

```
        }
    }
}
```

Scan D Flip-Flop With Auxiliary Clock

[Example 9-14](#) shows how to model a scan D flip-flop with one input and an auxiliary clock.

Example 9-14 Scan D Flip-Flop With Auxiliary Clock

```
cell(AUX_DFF1) {
    area : 12;
    pin(D) {
        direction : input;
        capacitance : 1;
        timing() {
            timing_type : setup_rising;
            related_pin : "CK";
        }
        timing() {
            timing_type : hold_rising;
            related_pin : "CK";
        }
        timing() {
            timing_type : setup_rising;
            related_pin : "IH";
        }
        timing() {
            timing_type : hold_rising;
            related_pin : "IH";
        }
    }
    pin(CK,IH,A,B) {
        direction : input;
        capacitance : 1;
    }
    pin(SI) {
        direction : input;
        capacitance : 1;
        prefer_tied : "0";
        timing() {
            timing_type : setup_falling;
            related_pin : "A";
        }
        timing() {
            timing_type : hold_falling;
            related_pin : "A";
        }
    }
    pin(Q) {
        direction : output;
        timing() {
```

```

        timing_type : rising_edge;
        related_pin : "CK IH";
    }
    timing() {
        timing_type : rising_edge;
        related_pin : "B";
    }
}
pin(QN) {
    direction : output;
    timing() {
        timing_type : rising_edge;
        related_pin : "CK IH";
    }
    timing() {
        timing_type : rising_edge;
        related_pin : "B";
    }
}
statetable( "C  TC  D  A  B  SI",   "IQ1  IQ2" ) {
    table : "\
/* C  TC D  A  B  SI           IQ1  IQ2  */ \
H  -  -  L  -  -  : -  -  : N  -, /* mast hold */ \
-  H  -  L  -  -  : -  -  : N  -, /* mast hold */ \
H  -  -  H  -  L/H : -  -  : L/H  -, /* scan mast */ \
-  H  -  H  -  L/H : -  -  : L/H  -, /* scan mast */ \
L  L  L/H L  -  -  : -  -  : L/H  -, /* D in mast */ \
L  L  -  H  -  -  : -  -  : X  -, /* both active */ \
H  -  -  -  L  -  : L/H  -  : -  L/H, /* slave loads */ \
-  H  -  -  L  -  : L/H  -  : -  L/H, /* slave loads */ \
L  L  -  -  -  -  : -  -  : -  N, /* slave loads */ \
-  -  -  -  H  -  : -  -  : -  N"; /* slave loads */
}
test_cell(){
    pin(D,CK){
        direction : input
    }
    pin(IH){
        direction : input;
        signal_type : "test_clock";
    }
    pin(SI){
        direction : input;
        signal_type : "test_scan_in";
    }
    pin(A){
        direction : input;
        signal_type : "test_scan_clock_a";
    }
    pin(B){
        direction : input;
        signal_type : "test_scan_clock_b";
    }
}

```

```
ff ("IQ","IQN") {
    next_state : "D";
    clocked_on : "CK";
}
pin(Q) {
    direction : output;
    function : "IQ";
    signal_type : "test_scan_out";
pin(QB) {
    direction : output;
    function : "IQN";
    signal_type : "test_scan_out_inverted";
}
}
```

10

Nonlinear Delay Model

Optimization tools use the timing parameters and environment attributes described in a technology library to calculate the timing delays for your designs. The timing parameters and environment attributes depend on the delay model. The delay model applies to all the cells in the library; you cannot mix delay models within a single library.

The CMOS nonlinear delay model (NLDM) uses lookup tables and interpolation to compute delays, and correlates well with multiple submicron delay modeling schemes. You define the CMOS NLDM with the following syntax:

```
delay_model : table_lookup ;
```

This chapter describes the timing parameters and environment attributes used by the CMOS NLDM, in the following sections:

- [Total Delay Equation](#)
- [Cell Delay](#)
- [Propagation Delay](#)
- [Transition Delay](#)
- [Connect Delay](#)
- [CMOS Nonlinear Delay Model Calculation](#)
- [Process, Voltage, and Temperature Scaling](#)

Total Delay Equation

In delay analysis, the total delay of a logic stage (delay between the input pin of a gate and the input pin of the next gate) is calculated.

The total delay has two major components: D_{cell} and D_C .

$$D_{total} = D_{cell} + D_C$$

D_{cell}

The delay of the gate, typically defined as the time interval between 50 percent of the input pin voltage and 50 percent output voltage. D_{cell} is computed based on the timing data provided. See “Cell Delay” on page 10-3 for a description of these computation methods.

D_C

The connect delay is either calculated with the `tree_type` attribute in the `operating_conditions` group and the selected `wire_load` model, or is read in from a delay feedback file as in the standard delay equation.

In the absence of a D_{cell} component, $D_{transition}$, which corresponds to the time required for the output pin to change state, is used. This is also referred to as the output ramp time.

$D_{transition}$

The time between two reference voltage levels on the output pin. For example, these levels might be 20 percent to 80 percent or 10 percent to 50 percent. Computing $D_{transition}$ involves performing table lookup and interpolation.

The NLDM supports two methods of computing D_{cell} . Although the two methods can be intermixed in a technology library, typically you specify the one method that correlates best to the characterized library data.

The two methods are

- Performing table lookup and interpolation in a cell delay table provided in the library
- Using the propagation and transition tables, following this equation:

$$D_{cell} = D_{propagation} + D_{transition}$$

The typical measurement for $D_{propagation}$ is the time from 50 percent input pin voltage until gate output just begins to switch—for example, when the 10 percent output voltage is reached. Thus, with a $D_{transition}$ value of 10 percent to 50 percent output voltage added to $D_{propagation}$, the result is a 50 percent input to 50 percent output cell delay.

If cell delay tables are provided for a timing arc, the total delay equation is

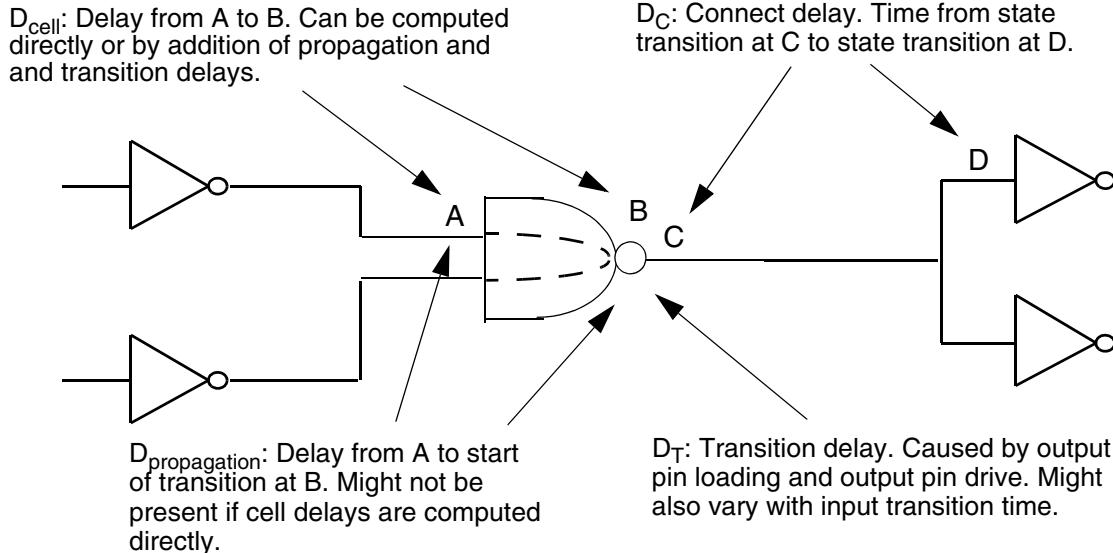
$$D_{\text{total}} = D_{\text{cell}} + D_C$$

If propagation delay tables are provided instead, the total delay equation is

$$D_{\text{total}} = D_{\text{propagation}} + D_{\text{transition}} + D_C$$

[Figure 10-1](#) shows the delay equation components.

Figure 10-1 Delay Equation Components for CMOS Nonlinear Delay Model



Cell Delay

Cell delay (D_{cell}) is typically defined as the time between reaching 50 percent of the input pin voltage to 50 percent of the output voltage. Cell delay is usually a function of both output loading and input transition time.

Two groups in the `timing` group define cell delay tables:

```
cell_rise
cell_fall
```

If you define cell delay tables for a timing arc, do not specify propagation delay tables for that arc.

For more information about how to define delay tables, see [Chapter 11, “Timing Arcs.”](#)

Propagation Delay

$D_{\text{propagation}}$ is the time from the input transition to completion of a specified percentage (for example, 20 percent) of the output transition. $D_{\text{propagation}}$ is often a function of output loading and input transition time.

Two groups in the `timing` group define propagation delay tables:

```
rise_propagation  
fall_propagation
```

If propagation delay tables are defined for a timing arc, cell delay tables must not be specified. The presence of propagation delay tables indicates that cell delays are computed by addition of the propagation and transition delays.

For more information about how to define delay tables, see [Chapter 11, “Timing Arcs.”](#)

Transition Delay

$D_{\text{transition}}$ is the time required for an output pin to change state. It is used as a term in the cell delay calculation if propagation tables are specified. After applicable transition degradation, it is also used to index into delay and transition tables at the next logic stage if the tables are indexed by `input_net_transition`. Transition delay can also be constrained as a design rule.

Computing $D_{\text{transition}}$ involves performing table lookup and interpolation. $D_{\text{transition}}$ is a function of capacitance at the output pin and can also be a function of input transition time in submicron technology.

Two groups are used to define transition delay tables:

```
rise_transition  
fall_transition
```

Transition tables must be specified for all delay arcs.

For more information about how to define delay tables, see [Chapter 11, “Timing Arcs.”](#)

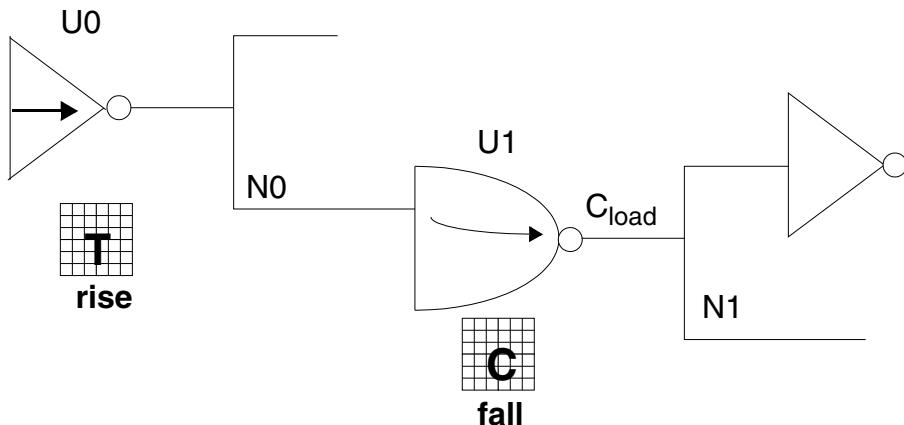
Connect Delay

D_C is the time it takes the voltage at an input pin to charge after the driving output pin has made a transition. This delay is also called the time-of-flight delay—the time it takes for a waveform to travel along a wire.

CMOS Nonlinear Delay Model Calculation

[Figure 10-2](#) is a schematic representation of the total delay through a cell. [Figure 10-3](#) displays the result of using a CMOS nonlinear delay model to determine delay through a cell. To determine the fall delay across the timing arc of cell U1 in [Figure 10-2](#), first examine the timing arc and determine whether propagation or cell delay tables are specified. In this case, a cell delay table (a two-dimensional table indexed by input transition time and total output capacitance) is specified.

Figure 10-2 Nonlinear Delay Calculation Schematic

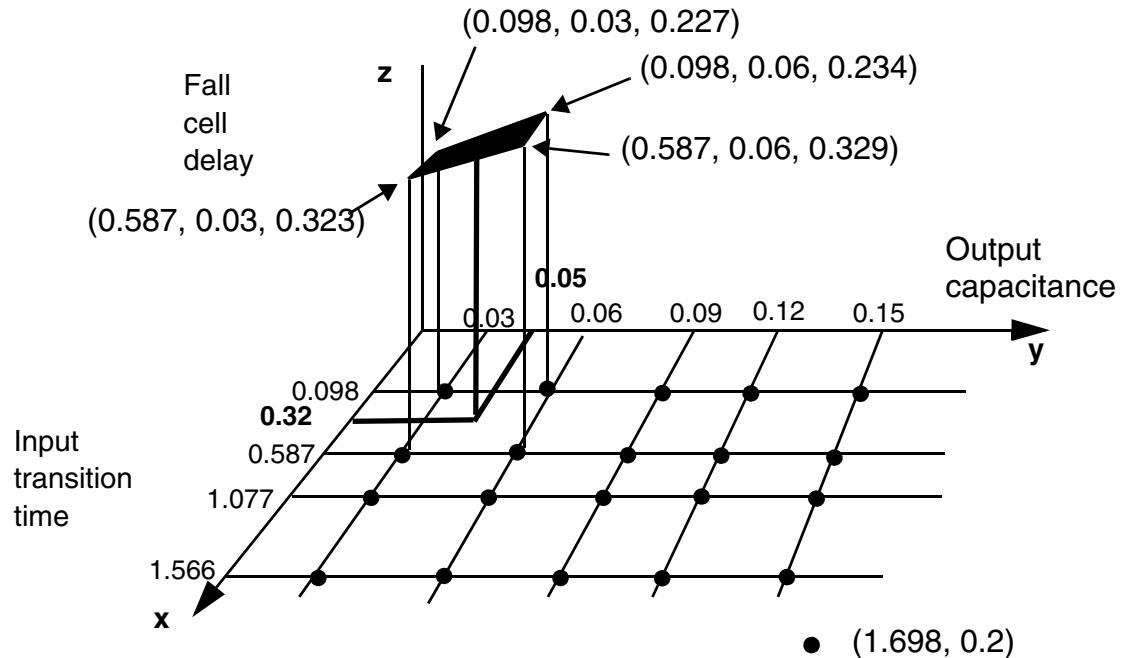


Input transition time is determined by evaluation of the transition delay at the previous gate, U0. Because the timing sense in U1 is negative unate, use the rise transition table for U0 to determine U1's input transition time. If multiple timing arcs are present at gate U0, the maximum of the rise transition values for those arcs is used as the input transition at U1.

Total output capacitance is calculated by addition of the capacitance introduced by the pins connected to net N1 and the capacitance contributed by the wire itself. For transition, cell, and propagation delays, the `tree_type` attribute of an `operating_conditions` group is not used in the total output capacitance calculation. For wire delay, wire capacitance is calculated by use of a `wire_load` model, but it can also be back-annotated.

Performing these calculations determines C_{load} to be 0.05 and the input transition time 0.32. You can now index into U1's fall cell delay table with these two values. Four neighboring table values are determined by examination of the table breakpoints.

For this example, the input transition time, 0.32, falls between the index values of 0.098 and 0.587. The output capacitance 0.05 falls between the index values of 0.03 and 0.06. The index values correspond to the x and y coordinates of the four neighboring table points as shown in [Figure 10-3](#). The corresponding z values are also shown in [Figure 10-3](#).

Figure 10-3 Result of Delay Calculation

You determine the approximation for the surface described by the three coordinates (x , y , z) in [Figure 10-3](#) by solving the A , B , C , and D coefficients of the following equation. Next, insert the coefficient values into the equation to determine which z -coordinate relates to the fall propagation delay (see [Example 10-1](#)).

$$Z = A + B*x + C*y + D*x*y$$

You can derive the coefficients A , B , C , and D with common mathematical methods, such as Gaussian elimination, as shown in [Example 10-1](#).

Example 10-1 Determining Coefficients and Solving for the Fall Cell Delay

$$\begin{aligned} 0.227 &= A + B * 0.098 + C * 0.03 + D * 0.098 * 0.03 \\ 0.234 &= A + B * 0.098 + C * 0.06 + D * 0.098 * 0.06 \\ 0.323 &= A + B * 0.587 + C * 0.03 + D * 0.587 * 0.03 \\ 0.329 &= A + B * 0.587 + C * 0.06 + D * 0.587 * 0.06 \end{aligned}$$

[Table 10-1](#) shows the coefficient values for the fall cell delay.

Table 10-1 Coefficient Values for Fall Cell Delay

Coefficient	Value
A	0.2006
B	0.1983
C	0.2399
D	0.0677

Insert the coefficient values and the x, y values that equal (0.32, 0.05) to solve for z (fall cell delay):

$$0.275 = 0.2006 + 0.1983 * 0.32 + 0.2399 * 0.05 + 0.0677 * 0.32 * 0.05$$

In two-dimensional interpolation, the dots in [Figure 10-3](#) represent points you defined in the library source file CMOS nonlinear delay model table. This information is extracted from a SPICE modeling simulation. The four points with heights shown on the z-axis are the neighboring points chosen for interpolation. The shaded area is the surface used for interpolation.

If the index in the table cannot cover the actual value for the dimension—for example, if `total_output_net_capacitance` is 0.2 in [Figure 10-3](#)—extrapolation is performed to extend the surface formed by the nearest four data points to cover it.

For example, if you have a data point (1.698, 0.2), use data points (1.077, 0.12), (1.077, 0.15), (1.566, 0.12), and (1.566, 0.15) to get a set of A, B, C, and D. Next, insert the data point (1.698, 0.2) to get its fall propagation delay, using the method shown in [Example 10-1](#).

For a one-dimensional table, the first and last line segments are extended to cover the data point if it falls outside the table index range.

Process, Voltage, and Temperature Scaling

When calculating total delay, Design Compiler scales each parameter of D_{total} individually. Each parameter has its own global parameters to model the effects on the nominal case of variation in process, temperature, and voltage.

The equation for a scaled version of an individual parameter is shown here:

$$P_{\text{scaled}} = P(1 + \Delta_{\text{process}} \times K_{\text{process}})(1 + \Delta_{\text{temp}} \times K_{\text{temp}})(1 + \Delta_{\text{voltage}} \times K_{\text{voltage}})$$

P

A parameter used in the total delay equation, such as cell, transition, or propagation table delay value; wire resistance or capacitance; or pin capacitance.

D_{process} , D_{temp} , and D_{voltage}

Represent the differences between the process, temperature, and voltage attributes of the prevailing `operating_conditions` group and the library's designated `nom_process`, `nom_temperature`, and `nom_voltage` attributes. For example,

$$D_{\text{process}} = \text{process} - \text{nom_process}$$

K_{process} , K_{temp} , and K_{voltage}

Represent the scaling factors of the individual components of each parameter.

Scaling factors for process, temperature, and voltage are called k-factors (scaling factors that begin with `k_`) and are defined at the `library` group level in the technology library. The Library Compiler tool assigns a 0 (zero) as the default for each k-factor attribute not defined in your library.

Nominal operating conditions and `operating_conditions` groups are also defined at the `library` group level. For a description of these statements, see the *Synopsys Logic Library Reference Manual*.

The k-factors apply to the table delay values, not to $D_{\text{transition}}$, $D_{\text{propagation}}$, or D_{cell} . The delay parameters are scaled before the delay is calculated. This means that post-scaling transition and capacitance values are used to index into delay tables.

In many technology modeling schemes, the overall delay (D_{total})—rather than individual components such as wire resistance, pin capacitance, or transition delay values—is scaled.

To scale D_{total} , specify the cell delay tables or propagation delay tables for timing arcs. Next, specify k-factors, to scale the cell table and propagation table values, and wire resistance, to achieve the overall delay scaling.

11

Timing Arcs

Timing arcs are divided into two major areas: timing delays (the actual circuit timing) and timing constraints (at the boundaries). This chapter explains the timing concepts and describes the `timing group` attributes for setting constraints and defining delay.

The following sections describe how to specify timing delays:

- [Understanding Timing Arcs](#)
- [Modeling Method Alternatives](#)
- [Defining the timing Group](#)
- [Describing Three-State Timing Arcs](#)
- [Describing Edge-Sensitive Timing Arcs](#)
- [Describing Preset and Clear Timing Arcs](#)
- [Describing Clock Insertion Delay](#)
- [Describing Intrinsic Delay](#)
- [Describing Transition Delay](#)
- [Modeling Load Dependency](#)
- [Describing Slope Sensitivity](#)
- [Describing State-Dependent Delays](#)

The following sections describe how to use timing constraints:

- [Setting Setup and Hold Constraints](#)
- [Setting Nonsequential Timing Constraints](#)
- [Setting Recovery and Removal Timing Constraints](#)
- [Setting No-Change Timing Constraints](#)
- [Setting Skew Constraints](#)
- [Setting Conditional Timing Constraints](#)

For additional information, see these sections:

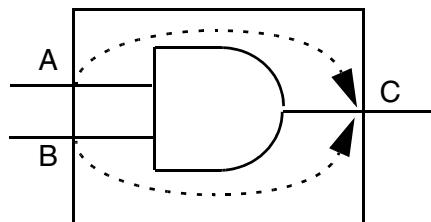
- [Impossible Transitions](#)
- [Examples of NLDM Libraries](#)
- [Describing a Transparent Latch Clock Model](#)
- [Checking Timing and Nonlinear Delay Data](#)
- [Driver Waveform Support](#)
- [Sensitization Support](#)
- [Phase-Locked Loop Support](#)

Understanding Timing Arcs

Timing arcs, along with netlist interconnect information, are the paths followed by the path tracer during path analysis. Each timing arc has a startpoint and an endpoint. The startpoint can be an input, output, or I/O pin. The endpoint is always an output pin or an I/O pin. The only exception is a constraint timing arc, such as a setup or hold constraint between two input pins.

[Figure 11-1](#) shows timing arcs AC and BC for an AND gate. All delay information in a library refers to an input-to-output pin pair or an output-to-output pin pair.

Figure 11-1 Timing Arcs



You must distinguish between combinational and sequential timing types, because they serve different purposes.

The Design Compiler tool uses combinational timing arc information to calculate the physical delays in timing propagation and to trace paths. The timing analyzer uses path-tracing arcs for circuit timing analysis.

The Design Compiler tool uses sequential timing arc information to determine rule-based design optimization constraints. See the *Synopsys Timing Constraints and Optimization User Guide* for more information about optimization constraints.

Combinational Timing Arcs

A combinational timing arc describes the timing characteristics of a combinational element. The timing arc is attached to an output pin, and the related pin is either an input or an output.

A combinational timing arc is of one of the following types:

- combinational
- combinational_rise
- combinational_fall
- three_state_disable

- three_state_disable_rise
- three_state_disable_fall
- three_state_enable
- three_state_enable_rise
- three_state_enable_fall

For information about describing combinational timing types, see “[timing Group Attributes](#)” on page 11-19.

Sequential Timing Arcs

Sequential timing arcs describe the timing characteristics of sequential elements. In descriptions of the relationship between a clock transition and data output (input to output), the timing arc is considered a *delay* arc. In descriptions of the relationship between a clock transition and data input (input to input), the timing arc is considered a *constraint* arc.

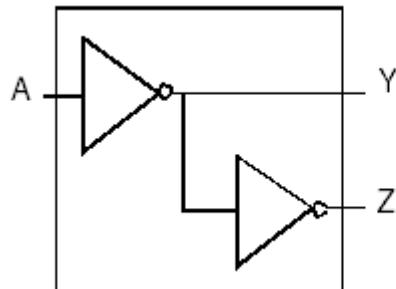
A sequential timing arc is of one of the following types:

- Edge-sensitive (rising_edge or falling_edge)
- Preset or clear
- Setup or hold (setup_rising, setup_falling, hold_rising, or hold_falling)
- Nonsequential setup or hold (non_seq_setup_rising, non_seq_setup_falling, non_seq_hold_rising, non_seq_hold_falling)
- Recovery or removal (recovery_rising, recovery_falling, removal_rising, or removal_falling)
- No change (nochange_high_high, nochange_high_low, nochange_low_high, nochange_low_low)

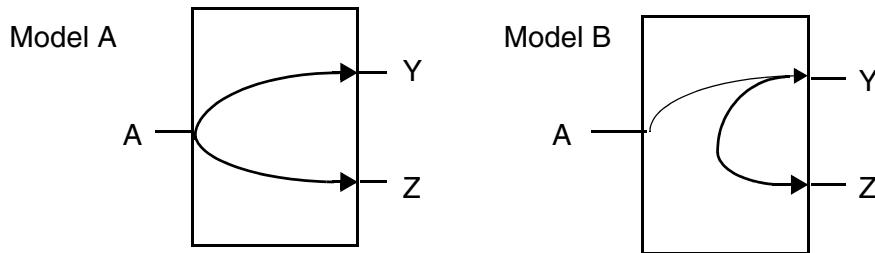
For information about describing sequential timing types, see “[timing Group Attributes](#)” on page 11-19.

Modeling Method Alternatives

Timing information for combinational cells such as the one in [Figure 11-2](#) can be modeled in one of two ways, as [Figure 11-3](#) shows.

Figure 11-2 Two-Inverter Cell

In [Figure 11-3](#), Model A defines two timing arcs. The first timing arc starts at primary input pin A and ends at primary output pin Y. The second timing arc starts at primary input pin A and ends at primary output pin Z. This is the simple case.

Figure 11-3 Two Modeling Techniques for Two-Inverter Cell

Model B for this cell also has two arcs but is more accurate than Model A. The first arc starts at pin A and ends at pin Y. This arc is modeled like the AY arc in Model A. The second arc is different; it starts with primary output Y and ends with primary output Z, modeling the effect the load on Y has on the delay on Z. Output-to-output timing arcs can be used in either combinational or sequential cells.

Defining the timing Group

The `timing` group contains information that downstream tools need to model timing arcs and trace paths. The `timing` group defines the timing arcs through a cell and the relationships between clock and data input signals.

The `timing` group describes

- Timing relationships between an input pin and an output pin
- Timing relationships between two output pins
- Timing arcs through a noncombinational element

- Setup and hold times on flip-flop or latch input
- Optionally, the names of the timing arcs

The `timing` group describes setup and hold information when the constraint information refers to an input-to-input pin pair.

The `timing` group is defined in the `pin` group. This is the syntax:

```
library (lib_name) {  
    cell (cell_name) {  
        pin (pin_name) {  
            timing () {  
                ... timing description ...  
            }  
        }  
    }  
}
```

Define the `timing` group in the `pin` group of the endpoint of the timing arc, as illustrated by pin C in [Figure 11-1 on page 11-3](#).

Note:

The Library Compiler tool allows you to define multiple timing arcs between two pins. However, other tools might have difficulty interpreting multiple timing arcs for pin pairs.

Naming Timing Arcs Using the `timing` Group

Within the `timing` group, you can identify the name or names of different timing arcs.

A single timing arc can occur between an identified pin and a single related pin identified with the `related_pin` attribute.

Multiple timing arcs can occur in many ways. The following list shows six possible multiple timing arcs. The following descriptive sections explain how you configure other possible multiple timing arcs:

- Between a single related pin and the identified multiple members of a bundle.
- Between multiple related pins and the identified multiple members of a bundle.
- Between a single related pin and the identified multiple bits on a bus.
- Between multiple related pins and the identified multiple bits of a bus.
- Between the identified multiple bits of a bus and the multiple pins of related bus pins (of a designated width).
- Between the internal pin and all the bits of the endpoint `bus` group.

The following sections provide descriptions and examples for various timing arcs.

Timing Arc Between a Single Pin and a Single Related Pin

Identify the timing arc that occurs between a single pin and a single related pin by entering a name in the `timing` group, as shown in the following example:

Example

```
cell (my_inverter) {
    ...
    pin (A) {
        direction : input;
        capacitance : 1;
    }
    pin (B) {
        direction : output
        function : "A'";
        timing (A_B) {
            related_pin : "A";
        }
        ...
    }/* end timing() */
}/* end pin B */
}/* end cell */
```

The timing arc is as follows:

From pin	To pin	Label
A	B	A_B

Timing Arcs Between a Pin and Multiple Related Pins

This section describes how to identify the timing arcs when a `timing` group is within a `pin` group and the timing arc has more than a single related pin.

Example

You identify the multiple timing arcs on a name list entered with the `timing` group as shown in the following example.

```
cell (my_and) {
    ...
    pin (A) {
        direction : input;
        capacitance : 1;
    }
    pin (B) {
```

```

        direction : input;
        capacitance : 2;
    }
    pin (C) {
        direction : output
        function : "A B";
        timing (A_C, B_C) {
            related_pin : "A B";
            ...
        }/* end timing() */
    }/* end pin B */
}/* end cell */

```

The timing arcs are as follows:

From pin	To pin	Label
A	C	A_C
B	C	B_C

Timing Arcs Between a Bundle and a Single Related Pin

When the `timing` group is within a `bundle` group that has several members with a single related pin, enter the names of the resulting multiple timing arcs in a name list in the `timing` group.

The Library Compiler tool assumes that the first name in the name list is the arc going from the related pin to the first pin in the bundle member list, the second name in the name list is the arc going from the related pin to the second pin in the bundle member list, and so on. See the following example.

Example

```

...
bundle (Q) {
    members (Q0, Q1, Q2, Q3);
    direction : output;
    function : "IQ";
    timing (G_Q0, G_Q1, G_Q2, G_Q3) {
        timing_type : rising_edge;
        related_pin : "G";
    }
}

```

If G is a pin, as opposed to another `bundle` group, the timing arcs are as follows:

From pin	To pin	Label
G	Q0	G_Q0
G	Q1	G_Q1
G	Q2	G_Q2
G	Q3	G_Q3

If G is another bundle of member size 4 and G0, G1, G2, and G3 are members of bundle G, the timing arcs are as follows:

From pin	To pin	Label
G0	Q0	G_Q0
G1	Q1	G_Q1
G2	Q2	G_Q2
G3	Q3	G_Q3

Timing Arcs Between a Bundle and Multiple Related Pins

When the `timing` group is within a `bundle` group that has several members, each having a corresponding related pin, enter the names of the resulting multiple timing arcs as a name list in the `timing` group.

The Library Compiler tool assumes that the first name in the name list is the arc going from the related pin to the first pin in the bundle member list, the second name in the name list is the arc going from the second related pin to the second pin in the bundle member list, and so on. See the following example.

Example

```
bundle (Q) {
    members (Q0, Q1, Q2, Q3);
    direction : output;
    function : "IQ";
    timing (G_Q0, H_Q0, G_Q1, H_Q1, G_Q2, H_Q2, G_Q3, H_Q3) {
        timing_type : rising_edge;
        related_pin : "G H";
    }
}
```

If G is a pin, as opposed to another `bundle` group, the timing arcs are as follows:

From pin	To pin	Label
G	Q0	G_Q0
H	Q0	H_Q0
G	Q1	G_Q1
H	Q1	H_Q1
G	Q2	G_Q2
H	Q2	H_Q2
G	Q3	G_Q3
H	Q3	H_Q3

If G was another bundle of member size 4 and G0, G1, G2, and G3 are members of bundle G, the timing arcs are as follows:

From pin	To pin	Label
G0	Q0	G_Q0
H	Q0	H_Q0
G1	Q1	G_Q1
H	Q1	H_Q1
G2	Q2	G_Q2
H	Q2	H_Q2
G3	Q3	G_Q3
H	Q3	H_Q3

The same rule applies if H is a size-4 bundle.

Timing Arcs Between a Bus and a Single Related Pin

This section describes how to identify the timing arcs created when a `timing` group is within a `bus` group that has several bits with the same single related pin. You identify the resulting multiple timing arcs by entering a name list with the `timing` group.

The Library Compiler tool assumes that the first name in the name list identifies the arc going from the related pin to the most significant bit (MSB) in the `bus` group, the second name in the name list identifies the arc going from the related pin to the second MSB in the `bus` group, and so on. See the following example.

Example

```
...
bus (X) {
/*assuming MSB is X[0] */
    bus_type : bus4;
    direction : output;
    capacitance : 1;
    pin (X[0:3]) {
        function : "B'";
        timing (B_X0, B_X1, B_X2, B_X3) {
            related_pin : "B";
        }
    }
}
```

If B is a pin, as opposed to another 4-bit bus, the timing arcs are as follows:

From pin	To pin	Label
B	X[0]	B_X0
B	X[1]	B_X1
B	X[2]	B_X2
B	X[3]	B_X3

If B is another 4-bit bus and B[0] is the MSB for bus B, the timing arcs are as follows:

From pin	To pin	Label
B[0]	X[0]	B_X0
B[1]	X[1]	B_X1
B[2]	X[2]	B_X2
B[3]	X[3]	B_X3

Timing Arcs Between a Bus and Multiple Related Pins

The Library Compiler tool assumes that the first name in the name list is the arc going from the first related pin to the most significant bit (MSB) in the `bus` group, the second name in the name list is the arc going from the second related pin to the second MSB in the `bus` group, and so on. See the following example:

This section describes the timing arcs created when a `timing` group is within a `bus` group that has several bits, where each bit has its own related pin. You identify the resulting multiple timing arcs by entering a name list with the `timing` group.

Example

```
bus (X) {
    /*assuming MSB is X[0] */
    bus_type : bus4;
    direction : output;
    capacitance : 1;
    pin (X[0:3]) {
        function : "B'";
        timing (B_X0, C_X0, B_X1, C_X1, B_X2, C_X2, B_X3, C_X3 ) {
            related_pin : "B' C";
        }
    }
}
```

If B and C are pins, as opposed to another 4-bit bus, the timing arcs are as follows:

From pin	To pin	Label
B	X[0]	B_X0
C	X[0]	C_X0
B	X[1]	B_X1
C	X[1]	C_X1
B	X[2]	B_X2
C	X[2]	C_X2
B	X[3]	B_X3
C	X[3]	C_X3

If B is another 4-bit bus and B[0] is the MSB for bus B, the timing arcs are as follows:

From pin	To pin	Label
B[0]	X[0]	B_X0
C	X[0]	C_X0
B[1]	X[1]	B_X1
C	X[1]	C_X1
B[2]	X[2]	B_X2
C	X[2]	C_X2
B[3]	X[3]	B_X3
C	X[3]	C_X3

The same rule applies if C is a 4-bit bus.

Delay Model

The `timing` groups are defined by the `timing group` attributes. The delay model determines the set of delay calculation attributes you specify in a `timing group`.

Cell delays are often modeled in a synthesis technology library, with an assortment of delay models. Synthesis tools use these models to predict cell delays during and after optimization. Net delays are estimated during synthesis on the basis of the wire load models provided in the technology library.

Synthesis tools support the CMOS nonlinear delay model (NLDM).

The NLDM is characterized by tables that define the timing arcs. To describe delay or constraint arcs with this model,

- Use the library-level `lu_table_template` group to define templates of common information to use in lookup tables.
- Use the templates and the timing groups described in this chapter to create lookup tables.

Lookup tables and their corresponding templates can be one-dimensional, two-dimensional, or three-dimensional. Delay arcs allow a maximum of two dimensions. Device degradation constraint tables allow only one dimension. Load-dependent constraint modeling requires three dimensions.

delay_model Attribute

To specify the delay model, use the `delay_model` attribute in the `library` group.

The `delay_model` attribute must be the first attribute in the `library` if a `technology` attribute is not present. Otherwise, it follows the `technology` attribute.

Example

```
library (demo) {  
    delay_model : table_lookup ;  
}
```

Defining the NLDM Template

Table templates store common table information that multiple lookup tables can use. A table template specifies the table parameters and the breakpoints for each axis. Assign each template a name so that lookup tables can refer to it.

lu_table_template Group

Define your lookup table templates in the `library` group.

Syntax

```
lu_table_template(name) {  
    variable_1 : value;  
    variable_2 : value;  
    variable_3 : value;
```

```

index_1 ("float, ... , float");
index_2 ("float, ... , float");
index_3 ("float, ... , float");
}

```

The Library Compiler tool includes a predefined lookup table template named scalar for tables having a single value.

Template Variables for Timing Delays

The table template specifying timing delays can have up to three variables (`variable_1`, `variable_2`, and `variable_3`). The variables indicate the parameters used to index into the lookup table along the first, second, and third table axes. The parameters are the input transition time of a constrained pin, the output net length and capacitance, and the output loading of a related pin.

The following is a list of the valid values (divided into sets) that you can assign to a table:

- Set 1:

```
input_net_transition
```

- Set 2:

```

total_output_net_capacitance
output_net_length
output_net_wire_cap
output_net_pin_cap

```

- Set 3:

```

related_out_total_output_net_capacitance
related_out_output_net_length
related_out_output_net_wire_cap
related_out_output_net_pin_cap

```

The values you can assign to the variables of a table specifying timing delay depend on whether the table is one-, two-, or three-dimensional.

For every table, the value you assign to a variable must be from a set different from the set from which you assign a value to the other variables. For example, if you want a two-dimensional table and you assign `variable_1` with the `input_net_transition` value from set 1, then you must assign `variable_2` with one of the values from set 2. [Table 11-1](#)

lists the combinations of values you can assign to the different variables for the varying dimensional tables specifying timing delays.

Table 11-1 Variable Values for Timing Delays

Template dimension	Variable_1	Variable_2	Variable_3
1	set1		
1	set2		
2	set1	set2	
2	set2	set1	
3	set1	set2	set3
3	set1	set3	set2
3	set2	set1	set3
3	set2	set3	set1
3	set3	set1	set2
3	set3	set2	set1

Template Variables for Load-Dependent Constraints

The table template specifying load-dependent constraints can have up to three variables (`variable_1`, `variable_2`, and `variable_3`). The variables indicate the parameters used to index into the lookup table along the first, second, and third table axes. The parameters are the input transition time of a constrained pin, the transition time of a related pin, and the output loading of a related pin.

The following is a list of the valid values (divided into sets) that you can assign to a table.

- Set 1:

```
constrained_pin_transition
```

- Set 2:

```
related_pin_transition
```

- Set 3:

```
related_out_total_output_net_capacitance
related_out_output_net_length
```

```
related_out_output_net_wire_cap
related_out_output_net_pin_cap
```

The values you can assign to the variables of a table specifying load-dependent constraints depend on whether the table is one-, two-, or three-dimensional.

For every table, the value you assign to a variable must be from a set different from the set from which you assign a value to the other variables. For example, if you want a two-dimensional table and you assign `variable_1` with the `constrained_pin_transition` value from set 1, then you must assign `variable_2` with one of the values from set 2.

Table 11-2 lists the combination of values you can assign to the different variables for the varying dimensional tables specifying load-dependent constraints.

Table 11-2 Variable Values for Load-Dependent Constraint Tables

Template dimension	Variable_1	Variable_2	Variable_3
1	set1	set2	set3
1	set2		
2	set1	set2	set3
2	set2	set1	
3	set1	set2	set3
3	set1	set3	set2
3	set2	set1	set3
3	set2	set3	set1
3	set3	set1	set2
3	set3	set2	set1

Template Breakpoints

The index statements define the breakpoints for an axis. The breakpoints defined by `index_1` correspond to the parameter values indicated by `variable_1`. The breakpoints defined by `index_2` correspond to the parameter values indicated by `variable_2`. The breakpoints defined by `index_3` correspond to the parameter values indicated by `variable_3`.

The index values are lists of floating-point numbers greater than or equal to 0.0. The values in the list must be in increasing order. The size of each dimension is determined by the number of floating-point numbers in the indexes.

You must define at least one `index_1` in the `lu_table_template` group. For a one-dimensional table, use only `variable_1`.

Creating Lookup Tables

The rules for specifying lookup tables apply to delay arcs as well as to constraints. “[Defining Delay Arcs With Lookup Tables](#)” on page 11-40 shows the groups to use as delay lookup tables. See the sections on the various constraints for the groups to use as constraint lookup tables.

This is the syntax for lookup table groups:

```
lu_table(name) {  
    index_1 ("float, ..., float");  
    index_2 ("float, ..., float");  
    index_3 ("float, ..., float");  
    values("float, ..., float", ..., "float, ..., float");  
}
```

These rules apply to lookup table groups:

- Each lookup table has an associated name for the `lu_table_template` it uses. The name of the template must be identical to the name defined in a library `lu_table_template` group.
- You can overwrite any or all of the indexes in a lookup table template, but the overwrite must occur before the actual definition of values.
- The delay value of the table is stored in a `values` attribute.
 - Transition table delay values must be 0.0 or greater. Propagation tables and cell tables can contain negative delay values.
 - In a one-dimensional table, represent the delay value as a list of `nindex_1` floating-point numbers.
 - In a two-dimensional table, represent the delay value as `nindex_1 x nindex_2` floating-point numbers.
 - If a table contains only one value, you can use the predefined scalar table template as the template for that timing arc. To use the scalar table template, place the string `scalar` in your lookup table group statement, as shown in [Example 11-4 on page 11-42](#).
- Each group of floating-point values enclosed in quotation marks represents a row in the table.

- In a one-dimensional table, the number of floating-point values in the group must equal `nindex_1`.
- In a two-dimensional table, the number of floating-point values in a group must equal `nindex_2` and the number of groups must equal `nindex_1`.
- In a three-dimensional table, the total number of groups is `nindex_1 × nindex_2` and each group contains as many floating-point numbers as `nindex_3`. In a three-dimensional table, the first group represents the value indexed by the $(1, 1, 1)$ to the $(1, 1, nindex_3)$ points in the index. The first `nindex_2` groups represent the value indexed by the $(1, 1, 1)$ to the $(1, nindex_2, nindex_3)$ points in the index. The rest of the groups are grouped in the same order.

[Example 11-4 on page 11-42](#) shows a library that uses the CMOS nonlinear delay model to describe the delay.

timing Group Attributes

[Table 11-3](#) shows the supported timing group attributes for NLDM.

Table 11-3 timing Group Attributes in NLDM

Purpose	Attribute or group
To specify a default timing arc	<code>default_timing</code>
To identify a timing arc startpoint	<code>related_pin</code> <code>related_bus_pins</code>
To describe a logical effect of input pin on output pin	<code>timing_sense</code>
To identify an arc as combinational or sequential	<code>timing_type</code>
To specify a propagation delay in total cell delay. (Used with transition delay)	<code>rise_propagation</code> <code>fall_propagation</code>
To specify a cell delay independent of transition delay	<code>cell_rise</code> <code>cell_fall</code>

Table 11-3 timing Group Attributes in NLDM (Continued)

Purpose	Attribute or group
To specify a retain delay within the delay arc	retaining_rise retaining_fall
To specify an output or I/O pin for load-dependency model	related_output_pin
To specify when a timing arc is active	mode

related_pin Simple Attribute

The `related_pin` attribute defines the pin or pins that are the startpoint of the timing arc. The primary use of `related_pin` is for multiple signals in ganged-logic timing. This attribute is a required component of all timing groups.

Example

```
pin (B) {
    direction : output ;
    function : "A'";
    timing () {
        related_pin : "A" ;
        ...
    }
}
```

You can use the `related_pin` attribute statement as a shortcut for defining two identical timing arcs for a cell. For example, for a 2-input NAND gate with identical delays from both input pins to the output pin, you need to define only one timing arc with two related pins, as shown in the following example.

```
pin (Z) {
    direction : output;
    function : "(A * B)'";
    timing () {
        related_pin : "A B" ;
        ... timing information ...
    }
}
```

When you use a bus name in a `related_pin` attribute statement, the bus members or the range of members is distributed across all members of the parent bus. In the following

example, the timing arcs described are from each element in bus A to each element in bus B: A(1) to B(1) and A(2) to B(2).

The width of the bus or range must be the same as the width of the parent bus.

```
bus (A) {
    bus_type : bus2;
    ...
}
bus (B) {
    bus_type : bus2;
    direction : output;
    function : "A'";
    timing () {
        related_pin : "A" ;
        ... timing information ...
    }
}
```

related_bus_pins Simple Attribute

The `related_bus_pins` attribute defines the pin or pins that are the startpoint of the timing arc. The primary use of `related_bus_pins` is for module generators.

Example

In this example, the timing arcs described are from each element in bus A to each element in bus B: A(1) to B(1), A(1) to B(2), A(1) to B(3), and so on. The widths of bus A and bus B do not need to be identical.

```
bus (A) {
    bus_type : bus2;
    ...
}
bus (B) {
    bus_type : bus4;
    direction : output ;
    function : "A";
    timing () {
        related_bus_pins : "A" ;
        ... timing information ...
    }
}
```

timing_sense Simple Attribute

The `timing_sense` attribute describes the way an input pin logically affects an output pin. A timing analyzer uses this attribute to track the polarity transition of an element during path analysis.

Example

```
timing () {  
    timing_sense : positive_unate;  
}
```

The Design Compiler tool typically derives the `timing_sense` value from the pin's logic function. For example, the value derived for an AND gate is `positive_unate`, the value for a NAND gate is `negative_unate`, and the value for an XOR gate is `non_unate`.

A function is *unate* if a rising (or falling) change on a positive unate input variable causes the output function variable to rise (or fall) or not change. A rising (or falling) change on a negative unate input variable causes the output function variable to fall (or rise) or not change. For a nonunate variable, further state information is required to determine the effects of a particular state transition.

Do not define the `timing_sense` value of a pin, except when you must override the derived value or you are characterizing a noncombinational gate such as a three-state component. For example, you might define the timing sense manually when you model multiple paths between an input pin and an output pin, as in an XOR gate.

It is possible that one path is positive unate while another is negative unate. In this case, the first timing arc gets a `positive_unate` designation and the second arc gets a `negative_unate` designation.

Note:

When `timing_sense` describes the transition edge used to calculate delay for the `three_state_enable` or `three_state_disable` pin, it has a meaning different from its traditional one. If a 1 value on the control pin of a three-state cell causes a Z value on the output pin, `timing_sense` is `positive_unate` for the `three_state_disable` timing arc and `negative_unate` for the `three_state_enable` timing arc. If a 0 value on the control pin of a three-state cell causes a Z value on the output pin, `timing_sense` is `negative_unate` for the `three_state_disable` timing arc and `positive_unate` for the `three_state_enable` timing arc.

If a `related_pin` is an output pin, you must define `timing_sense` for that pin.

timing_type Simple Attribute

The `timing_type` attribute distinguishes between combinational and sequential cells, by defining the type of timing arc. If this attribute is not assigned, the cell is considered combinational.

You must distinguish between combinational and sequential timing types, because each type serves a different purpose.

The Design Compiler tool uses the combinational timing arcs information to calculate the physical delays in timing propagation and to trace paths. The timing analyzer uses path tracing arcs for circuit timing analysis.

The Design Compiler tool uses the sequential timing arcs information to determine rule-based design optimization constraints. More information about optimization constraints is available in the Design Compiler documentation.

Example

```
timing () {
    timing_type : rising_edge;
}
```

Table 11-4 shows the valid values for the `timing_type` attribute:

Table 11-4 Valid Values for the timing_type Attribute

Value	Cells where used	Function
combinational	combinational	Use this timing type value to describe 1-to-0 or 0-to-1 timing arcs. This is the default when there is no <code>timing_type</code> attribute in the <code>timing</code> group.
combinational_rise	combinational	Use these timing type values to describe 0-to-1 or 1-to-0 timing arcs, respectively.
combinational_fall		
three_state_disable	combinational	Use these values to describe three-state output pins.
three_state_enable		
three_state_disable_rise		
three_state_disable_fall		
three_state_enable_rise		
three_state_enable_fall		
rising_edge	sequential	Use these values to specify which edge of the input signal causes a transition on the output pin.
falling_edge		
preset	sequential	Use these values to affect fall or rise arrival time on a timing arc's endpoint pin.
clear		
hold_rising	sequential	Use these values to designate the edge of the related pin for the hold check.
hold_falling		
setup_rising	sequential	Use these values to designate the edge of the related pin for the setup check on clocked elements.
setup_falling		

Table 11-4 Valid Values for the timing_type Attribute (Continued)

Value	Cells where used	Function
recovery_rising recovery_falling	sequential	Use these values to designate the edge of the related pin for the recovery time check.
skew_rising skew_falling	sequential	Use these values to specify constraints defining the maximum separation between two clock signals. Use these values for simulation only.
removal_rising removal_falling	sequential	Use these values to designate the edge that defines the removal time.
minimum_period	sequential	Use these values to model minimum period constraints for an input pin.
min_pulse_width	combinational or sequential	Use these values to model pulse width constraints for an input pin.
max_clock_tree_path min_clock_tree_path	sequential	Use these values to define the minimum and maximum clock tree path constraints.
non_seq_setup_rising non_seq_setup_falling non_seq_hold_rising non_seq_hold_falling	combinational or sequential	Use these values to specify setup and hold constraints on data arrival unrelated to clock signals.
nochange_high_high nochange_high_low nochange_low_high nochange_low_low	sequential	Use these values to specify no-change constraints on signal pulses with relation to clock pulses.

The following sections show the `timing_type` attribute values for the following types of timing arcs:

- Combinational
- Sequential
- Nonsequential
- No-change

Values for Combinational Timing Arcs

The timing type and timing sense define the signal propagation pattern. The default timing type is combinational.

Timing type	Timing sense		
	positive_unate	negative_unate	non_unate
combinational	R->R,F->F	R->F,F->R	{R,F}->{R,F}
combinational_rise	R->R	F->R	{R,F}->R
combinational_fall	F->F	R->F	{R,F}->F
three_state_disable	R->{0Z,1Z}	F->{0Z,1Z}	{R,F}->{0Z,1Z}
three_state_enable	R->{Z0,Z1}	F->{Z0,Z1}	{R,F}->{Z0,Z1}
three_state_disable_rise	R->0Z	F->0Z	{R,F}->0Z
three_state_disable_fall	R->1Z	F->1Z	{R,F}->1Z
three_state_enable_rise	R->Z1	F->Z1	{R,F}->Z1
three_state_enable_fall	R->Z0	F->Z0	{R,F}->Z0

Values for Sequential Timing Arcs

You use sequential timing arcs to model the timing requirements for sequential cells.

`rising_edge`

Identifies a timing arc whose output pin is sensitive to a rising signal at the input pin.

`falling_edge`

Identifies a timing arc whose output pin is sensitive to a falling signal at the input pin.

`preset`

Preset arcs affect only the rise arrival time of the arc's endpoint pin. A preset arc implies that you are asserting a logic 1 on the output pin when the designated related pin is asserted.

`clear`

Clear arcs affect only the fall arrival time of the arc's endpoint pin. A clear arc implies that you are asserting a logic 0 on the output pin when the designated related pin is asserted.

`hold_rising`

Designates the rising edge of the related pin for the hold check.

`hold_falling`

Designates the falling edge of the related pin for the hold check.

`setup_rising`

Designates the rising edge of the related pin for the setup check on clocked elements.

`setup_falling`

Designates the falling edge of the related pin for the setup check on clocked elements.

`recovery_rising`

Uses the rising edge of the related pin for the recovery time check. The clock is rising-edge-triggered.

`recovery_falling`

Uses the falling edge of the related pin for the recovery time check. The clock is falling-edge-triggered.

`skew_rising`

The timing constraint interval is measured from the rising edge of the reference pin (specified in `related_pin`) to a transition edge of the parent pin in the `timing` group. The `rise_constraint` value is the maximum skew time between the reference pin rising and the parent pin rising. The `fall_constraint` value is the maximum skew time between the reference pin falling and the parent pin falling.

`skew_falling`

The timing constraint interval is measured from the falling edge of the reference pin (specified in `related_pin`) to a transition edge of the parent pin in the `timing` group. The `rise_constraint` value is the maximum skew time between the reference pin falling and the parent pin rising. The `fall_constraint` value is the maximum skew time between the reference pin falling and the parent pin falling.

`removal_rising`

Used when the cell is a low-enable latch or a rising-edge-triggered flip-flop. For active-low asynchronous control signals, define the removal time with the `rise_constraint` group. For active-high asynchronous control signals, define the removal time with the `fall_constraint` group.

removal_falling

Used when the cell is a high-enable latch or a falling-edge-triggered flip-flop. For active-low asynchronous control signals, define the removal time with the `rise_constraint` group. For active-high asynchronous control signals, define the removal time with the `fall_constraint` group.

min_pulse_width

This value, together with the `minimum_period` value, lets you specify the minimum pulse width for a clock pin. The timing check is performed on the pin itself, so the related pin should be the same. You can also include rise and fall constraints, as with other timing checks.

Besides scalar values, table-based minimum pulse width is supported. For an example, see “A Library With `timing_type` Statements” ([Example 11-11 on page 11-76](#)).

minimum_period

This value, together with the `min_pulse_width` value, lets you specify the minimum pulse width for a clock pin. The timing check is performed on the pin itself, so the related pin should be the same. You can also include rise and fall constraints as with other timing checks.

max_clock_tree_path

Used in `timing` groups under a clock pin. Defines the maximum clock tree path constraint. The Library Compiler tool checks that you have not specified a `related_pin` attribute.

min_clock_tree_path

Used in `timing` groups under a clock pin. Defines the minimum clock tree path constraint. The Library Compiler tool checks that you have not specified a `related_pin` attribute.

Values for Nonsequential Timing Arcs

In some nonsequential cells, the setup and hold timing constraints are specified on the data pin with a nonclock pin as the related pin. The signal of a pin must be stable for a specified period of time before and after another pin of the same cell range state for the cell to function as expected.

non_seq_setup_rising

Defines (with `non_seq_setup_falling`) the timing arcs used for setup checks between pins with nonsequential behavior. The related pin in a timing arc is used for the timing check. The `_rising` designation tells Design Compiler that the rising edge of the related pin is active for the setup check.

`non_seq_setup_falling`

Defines (with `non_seq_setup_rising`) the timing arcs used for setup checks between pins with nonsequential behavior. The related pin in a timing arc is used for the timing check. The `_falling` designation tells Design Compiler that the falling edge of the related pin is active for the setup check.

`non_seq_hold_rising`

Defines (with `non_seq_hold_falling`) the timing arcs used for hold checks between pins with nonsequential behavior. The related pin in a timing arc is used for the timing check. The `_rising` designation tells Design Compiler that the rising edge of the related pin is active for the hold check.

`non_seq_hold_falling`

Defines (with `non_seq_hold_rising`) the timing arcs used for hold checks between pins with nonsequential behavior. The related pin in a timing arc is used for the timing check. The `_falling` designation tells Design Compiler that the falling edge of the related pin is active for the hold check.

Values for No-Change Timing Arcs

You use no-change timing arcs to model the timing requirement for latch devices with latch-enable signals. The four no-change timing types define the pulse waveforms of both the constrained signal and the related signal in NLDM. The information is used in static timing verification during synthesis.

`nochange_high_high`

Indicates a positive pulse on the constrained pin and a positive pulse on the related pin.

`nochange_high_low`

Indicates a positive pulse on the constrained pin and a negative pulse on the related pin.

`nochange_low_high`

Indicates a negative pulse on the constrained pin and a positive pulse on the related pin.

`nochange_low_low`

Indicates a negative pulse on the constrained pin and a negative pulse on the related pin.

mode Complex Attribute

You define the `mode` attribute within a `timing` group. A `mode` attribute pertains to an individual timing arc. The timing arc is active when `mode` is instantiated with a name and a value. You can specify multiple instances of the `mode` attribute, but only one instance for each timing arc.

Syntax

```
mode (mode_name, mode_value);
```

The Library Compiler tool issues an error message if the *mode_name* and *mode_value* strings are not already defined by a `mode_definition` group in the cell.

Example

```
timing() {
    mode(rw, read);
}
```

[Example 11-1](#) shows a `mode` instance description.

Example 11-1 A mode Instance Description

```
pin(my_outpin) {
    direction : output;
    timing() {
        related_pin : b;
        timing_sense : non_unate;
        mode(rw, read);
        cell_rise(delay3x3) {
            values("1.1, 1.2, 1.3", "2.0, 3.0, 4.0", "2.5, 3.5, 4.5");
        }
        rise_transition(delay3x3) {
            values("1.0, 1.1, 1.2", "1.5, 1.8, 2.0", "2.5, 3.0, 3.5");
        }
        cell_fall(delay3x3) {
            values("1.1, 1.2, 1.3", "2.0, 3.0, 4.0", "2.5, 3.5, 4.5");
        }
        fall_transition(delay3x3) {
            values("1.0, 1.1, 1.2", "1.5, 1.8, 2.0", "2.5, 3.0, 3.5");
        }
    }
}
```

[Example 11-2](#) shows multiple `mode` descriptions.

Example 11-2 Multiple mode Descriptions

```
library (MODE_EXAMPLE) {
    delay_model           : "table_lookup";
    time_unit              : "1ns";
    voltage_unit           : "1V";
    current_unit           : "1mA";
    pulling_resistance_unit : "1kohm";
    leakage_power_unit     : "1nW";
    capacitive_load_unit   : (1, pf);
    nom_process             : 1.0;
    nom_voltage              : 1.0;
    nom_temperature          : 125.0;
    slew_lower_threshold_pct_rise : 10 ;
    slew_upper_threshold_pct_rise : 90 ;
    input_threshold_pct_fall   : 50 ;
```

```

output_threshold_pct_fall      : 50 ;
input_threshold_pct_rise       : 50 ;
output_threshold_pct_rise      : 50 ;
slew_lower_threshold_pct_fall : 10 ;
slew_upper_threshold_pct_fall : 90 ;
slew_derate_from_library      : 1.0 ;
cell_(<mode_example>) {
    mode_definition(RAM_MODE) {
        mode_value(MODE_1) {
        }
        mode_value(MODE_2) {
        }
        mode_value(MODE_3) {
        }
        mode_value(MODE_4) {
        }
    }
    interface_timing : true;
    dont_use        : true;
    dont_touch      : true;
    pin(Q) {
        direction       : output;
        max_capacitance : 2.0;
        three_state     : "!OE";
        timing() {
            related_pin   : "CK";
            timing_sense  : non_unate;
            timing_type   : rising_edge;
            mode(RAM_MODE,"MODE_1 MODE_2");
            cell_rise(scalar) {
                values( " 0.0 ");
            }
            cell_fall(scalar) {
                values( " 0.0 ");
            }
            rise_transition(scalar) {
                values( " 0.0 ");
            }
            fall_transition(scalar) {
                values( " 0.0 ");
            }
        }
        timing() {
            related_pin   : "OE";
            timing_sense  : positive_unate;
            timing_type   : three_state_enable;
            mode(RAM_MODE, " MODE_2 MODE_3");
            cell_rise(scalar) {
                values( " 0.0 ");
            }
            cell_fall(scalar) {
                values( " 0.0 ");
            }
        }
    }
}

```

```
    rise_transition(scalar) {
        values( " 0.0 ");
    }
    fall_transition(scalar) {
        values( " 0.0 ");
    }
}
timing() {
    related_pin      : "OE";
    timing_sense     : negative_unate;
    timing_type      : three_state_disable;
    mode(RAM_MODE, MODE_3);
    cell_rise(scalar) {
        values( " 0.0 ");
    }
    cell_fall(scalar) {
        values( " 0.0 ");
    }
    rise_transition(scalar) {
        values( " 0.0 ");
    }
    fall_transition(scalar) {
        values( " 0.0 ");
    }
}
pin(A) {
    direction         : input;
    capacitance       : 1.0;
    max_transition    : 2.0;
    timing() {
        timing_type      : setup_rising;
        related_pin      : "CK";
        mode(RAM_MODE, MODE_2);
        rise_constraint(scalar) {
            values( " 0.0 ");
        }
        fall_constraint(scalar) {
            values( " 0.0 ");
        }
    }
    timing() {
        timing_type      : hold_rising;
        related_pin      : "CK";
        mode(RAM_MODE, MODE_2);
        rise_constraint(scalar) {
            values( " 0.0 ");
        }
        fall_constraint(scalar) {
            values( " 0.0 ");
        }
    }
}
```

```
pin(OE) {
    direction      : input;
    capacitance   : 1.0;
    max_transition : 2.0;
}
pin(CS) {
    direction      : input;
    capacitance   : 1.0;
    max_transition : 2.0;
    timing() {
        timing_type      : setup_rising;
        related_pin       : "CK";
        mode(RAM_MODE, MODE_1);
        rise_constraint(scalar) {
            values( " 0.0 ");
        }
        fall_constraint(scalar) {
            values( " 0.0 ");
        }
    }
    timing() {
        timing_type      : hold_rising;
        related_pin       : "CK";
        mode(RAM_MODE, MODE_1);
        rise_constraint(scalar) {
            values( " 0.0 ");
        }
        fall_constraint(scalar) {
            values( " 0.0 ");
        }
    }
}
pin(CK) {
    timing() {
        timing_type : "min_pulse_width";
        related_pin : "CK";
        mode(RAM_MODE , MODE_4);
        fall_constraint(scalar) {
            values( " 0.0 ");
        }
        rise_constraint(scalar) {
            values( " 0.0 ");
        }
    }
    timing() {
        timing_type : "minimum_period";
        related_pin : "CK";
        mode(RAM_MODE , MODE_4);
        rise_constraint(scalar) {
            values( " 0.0 ");
        }
        fall_constraint(scalar) {
            values( " 0.0 ");
        }
    }
}
```

```

        }
    }
    clock          : true;
    direction      : input;
    capacitance    : 1.0;
    max_transition : 1.0;
}
cell_leakage_power : 0.0;
}
}

```

Describing Three-State Timing Arcs

Three-state arcs describe a three-state output pin in a cell.

Describing Three-State-Disable Timing Arcs

To designate a three-state-disable timing arc when defining a three-state pin,

1. Assign `related_pin` to the enable pin of the three-state function.
2. Define the 0-to-Z propagation time with the `cell_rise` or `rise_transition` statement.
3. Define the 1-to-Z propagation time with the `cell_fall` or `fall_transition` statement.
4. Include the `timing_type:three_state_disable` statement.

Example

```

timing () {
    related_pin : "OE" ;
    timing_type : three_state_disable ;
/* 0 to Z modeling */
    cell_rise(scalar) {
        values( " 0.0 ");
    }
    rise_transition(scalar) {
        values( " 0.0 ");
    }

/* 1 to Z modeling */
    cell_fall(scalar) {
        values( " 0.0 ");
    }
    fall_transition(scalar) {
        values( " 0.0 ");
    }
}

```

Note:

The `timing_sense` attribute, which describes the transition edge used to calculate delay for a timing arc, has a nontraditional meaning when it is included in a `timing` group that also contains a `three_state_disable` attribute. See “[timing_sense Simple Attribute](#)” on page 11-21 for more information.

Describing Three-State-Enable Timing Arcs

To designate a three-state-enable timing arc when defining a three-state pin,

1. Assign `related_pin` to the enable pin of the three-state function.
2. Define the Z-to-1 propagation time with the `cell_rise` or `rise_transition` group.
3. Define the Z-to-0 propagation time with the `cell_fall` or `fall_transition` group.
4. Include the `timing_type : three_state_enable` statement.

Example

```
timing () {
    related_pin : "OE" ;
    timing_type : three_state_enable ;

    /* 0 to Z modeling */
    cell_rise(scalar) {
        values( " 0.0 ");
    }
    rise_transition(scalar) {
        values( " 0.0 ");
    }
    /* 1 to Z modeling */
    cell_fall(scalar) {
        values( " 0.0 ");
    }
    fall_transition(scalar) {
        values( " 0.0 ");
    }
}
```

Note:

The `timing_sense` attribute that describes the transition edge used to calculate delay for a timing arc has a nontraditional meaning when it is included in a `timing` group that also contains a `three_state_enable` attribute. See “[timing_sense Simple Attribute](#)” on page 11-21 for more information.

[Example 11-3](#) shows a three-state cell with both disable and enable timing arcs.

Example 11-3 Three-State Cell With Disable and Enable Timing Arcs

```
library(example) {
```

```

date : "January 14, 2015";
revision : 2015.01;
technology (cmos);
...
cell(TRI_INV2) {
    area : 3;
    pin(A) {
        direction : input;
        capacitance : 2;
    }
    pin(E) {
        direction : input;
        capacitance : 2;
    }
    pin(Z) {
        direction : output;
        function : "A'";
        three_state : "E'";
        timing() {
            related_pin : "A";
        }
        timing() {
            timing_type : three_state_enable;
            related_pin : "E";
        }
        timing() {
            timing_type : three_state_disable;
            related_pin : "E";
        }
    }
}
}
}

```

Describing Edge-Sensitive Timing Arcs

Edge-sensitive timing arcs, such as the arc from the clock on a flip-flop, are identified by the following values of the `timing_type` attribute in the `timing` group.

`rising_edge`

Identifies a timing arc whose output pin is sensitive to a rising signal at the input pin.

`falling_edge`

Identifies a timing arc whose output pin is sensitive to a falling signal at the input pin.

These arcs are path-traced; the path tracer propagates only the active edge (rise or fall) path values along the timing arc.

See “[timing_type Simple Attribute](#)” on page 11-22 for information about the `timing_type` attribute.

The following example shows the timing arc for the QN pin of a JK flip-flop.

Example

```
pin(QN) {
    direction : output ;
    function : "IQN" ;
    timing() {
        related_pin : "CP" ;
        timing_type : rising_edge ;
    }
}
```

The QN pin makes a transition after the clock signal rises.

Describing Preset and Clear Timing Arcs

In normal operation, preset arcs affect only the rise arrival time on the arc's endpoint pin and clear arcs affect only the fall arrival time. You can use these timing arcs for asynchronous reset and clear pins on a flip-flop or a level-sensitive latch. Accordingly, a rise time must be defined for a preset arc and a fall time must be defined for a clear arc. The rise time for a preset arc is defined in the `cell_rise` group. The fall timing for a clear arc is defined similarly. Generally, fall timing for preset arcs and rise timing for clear arcs do not exist.

In some cases, a transition on a preset pin can cause a falling transition at the arc's endpoint pin and a transition on a clear pin can cause a rising transition. For example, for the preset arc, consider what happens when a preset signal is de-asserted on a level-sensitive latch while the latch input is at logic 0 and the latch is in its transparent mode. In such a case, the output of the latch is a falling transition caused by the de-assertion of the preset signal. For the clear arc case, consider what happens when the clear signal on an edge-triggered flip-flop is de-asserted while the preset signal is still asserted and the clear signal dominates the preset signal. In such a case, the flip-flop output is a rising transition caused by the de-assertion of the clear signal.

In normal operation, the Design Compiler and PrimeTime tools ignore paths with a falling transition at a preset arc endpoint pin and paths with a rising transition at a clear arc endpoint pin. This is done without a warning message. To force the PrimeTime tool to consider such paths, a fall timing must be defined for a preset arc and a rise timing must be defined for a clear arc. However, these rise and fall timing values must be defined only for a case similar to one of the two examples described in this section.

Describing Preset Arcs

Preset arcs affect only the rise arrival time of the arc's endpoint pin. A preset arc means that you are asserting a logic 1 on the output pin when the designated `related_pin` is asserted.

To designate a preset arc,

1. Select the preset value for the `timing_type` attribute.

```
timing_type : preset;
```

2. Assign the appropriate value to the `timing_sense` attribute. The valid values are `positive_unate`

Indicates that the rise arrival time of the arc's source pin is used to calculate the arc's delay. This calculation produces the rise arrival time on the arc's endpoint pin. In the case of slope delays, the source pin's rise transition time is added to the arc's delay. The source pin is active-high.

`negative_unate`

Indicates that the fall arrival time of the arc's source pin is used to calculate the arc's delay. This calculation produces the rise arrival time on the arc's endpoint pin. In the case of slope delays, the source pin's fall transition time is added to the arc's delay. The source pin is active-low.

`non_unate`

Indicates that the maximum of the rise and fall arrival times of the arc's source pin is used to calculate the arc's delay. This calculation produces the maximum arrival time on the arc's endpoint pin. In the case of slope delays, the maximum of the source pin's rise and fall transition times is added to the arc's delay.

```
timing_sense : negative_unate;
```

If you are specifying a preset arc for a cell that has both clear and preset pins, define the fall time for the preset arc to accurately model the dynamic timing behavior (for simulation tools) during the time when both clear and preset signals are active.

Describing Clear Arcs

Clear arcs affect only the fall arrival time of the arc's endpoint pin. A clear arc means that you are asserting a logic 0 on the output pin when the designated `related_pin` is asserted.

In normal operation, a clear arc causes a falling transition at the output pin. In some cases, however, a transition on a clear pin can cause a rising transition at the output pin. For example, consider what happens when CDN is active-low clear and SDN is active-low preset:

```
SDN = 0
CDN transition from 0 to 1
```

The result is

Q transition from 0 to 1

In such a case, you can define a rise delay for the clear arc.

To designate a clear arc,

1. Select the clear value for the `timing_type` attribute.

```
timing_type : clear;
```

2. Assign the appropriate value to the `timing_sense` attribute. These are valid values:
`positive_unate`

Indicates that the fall arrival time of the arc's source pin is used to calculate the arc's delay. This calculation produces the fall arrival time on the arc's endpoint pin. In the case of slope delays, the source pin's fall transition time is added to the arc's delay. The source pin is active-low.

`negative_unate`

Indicates that the rise arrival time of the arc's source pin is used to calculate the arc's delay. This calculation produces the fall arrival time on the arc's endpoint pin. In the case of slope delays, the source pin's rise transition time is added to the arc's delay. The source pin is active-high.

`non_unate`

Indicates that the maximum of the rise and fall arrival times of the arc's source pin is used in calculating the arc's delay. This calculation produces the maximum fall arrival time on the arc's endpoint pin. In the case of slope delays, the maximum of the source pin's rise and fall transition times is added to the arc's delay.

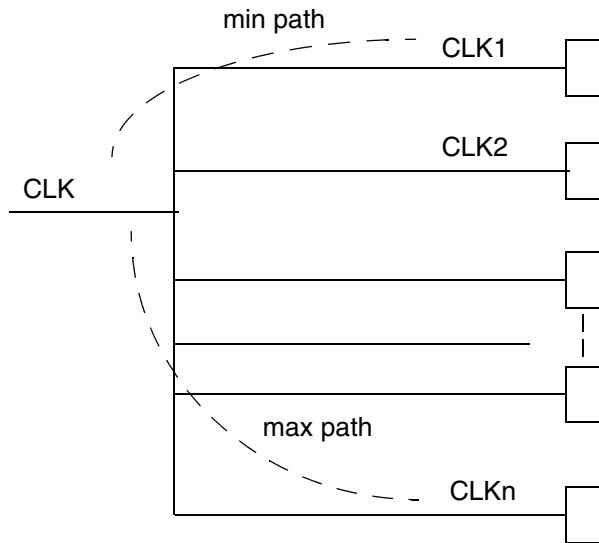
```
timing_sense : positive_unate;
```

If you are specifying a clear arc for a cell that has both clear and preset pins, define the rise time for the clear arc to accurately model the dynamic timing behavior (for simulation tools) when both clear and preset signals are active.

Describing Clock Insertion Delay

Arrival timing paths are the timing paths from an input clock pin to the clock pins that are internal to a cell. The arrival timing paths describe the minimum and the maximum timing constraint for a pin driving an internal clock tree for each input transition, as shown in [Figure 11-4](#).

Figure 11-4 Minimum and Maximum Clock Tree Paths



The `max_clock_tree_path` and `min_clock_tree_path` attributes let you define the maximum and minimum clock tree path constraints.

The clock tree path for any one clock can have up to eight values depending on the unateness of the pins and the fastest and slowest paths.

You can use lookup tables to model the cell delays. Lookup tables are indexed only by the input transition time of the clock.

For timing groups whose `timing_sense` attribute is set to `non_unate` and whose only variable is `input_net_transition`, use pairs of lookup tables to model both positive unate and negative unate.

Describing Intrinsic Delay

The intrinsic delay of an element is the zero-load (fixed) component of the total delay equation. Intrinsic delay attributes have different meanings, depending on whether they are for an input or an output pin.

When describing an output pin, the intrinsic delay attributes define the fixed delay from input to output pin. These values are used to calculate the intrinsic delay of the total delay equation.

When describing an input pin, such as in a setup or hold timing arc, intrinsic attributes define the timing requirements for that pin. Timing constraints are not used in the delay equation. The description of intrinsic delay is inherent in the lookup tables you create.

Describing Transition Delay

The transition delay of an element is the time it takes the driving pin to change state. Transition delay attributes represent the resistance encountered in making logic transitions.

The components of the total delay calculation depend on the timing delay model used. Include the transition delay attributes that apply to the delay model you are using.

Transition time is the time it takes for an output signal to make a transition between the high and low logic states. It is computed by table lookup and interpolation. Transition delay is a function of capacitance at the output pin and input transition time.

Defining Delay Arcs With Lookup Tables

These `timing` group attributes provide valid lookup tables for delay arcs:

- `cell_rise`
- `cell_fall`
- `rise_propagation`
- `fall_propagation`
- `retaining_rise`
- `retaining_fall`
- `retain_rise_slew`
- `retain_fall_slew`

Note:

For `timing` groups with timing type clear, only fall groups are valid. For `timing` groups with timing type preset, only rise groups are valid.

There are two methods for defining delay arcs. Choose the method that best fits your library data characterization. See the *Synopsys Logic Library Reference Manual* for details about the way delays are computed with lookup tables.

Method 1

To specify cell delay independently of transition delay, use one of these `timing` group attributes as your lookup table:

- `cell_rise`
- `cell_fall`

Method 2

To specify transition delay as a term in the total cell delay, use one of these `timing` group attributes as your lookup table:

- `rise_propagation`
- `fall_propagation`

`cell_rise` and `cell_fall` Groups

The cell delay is usually a function of both output loading and input transition time.

These groups define cell delay lookup tables (independently of transition delay) in CMOS nonlinear timing models. If you define cell delay tables for a timing arc, you cannot specify propagation delay tables for that arc.

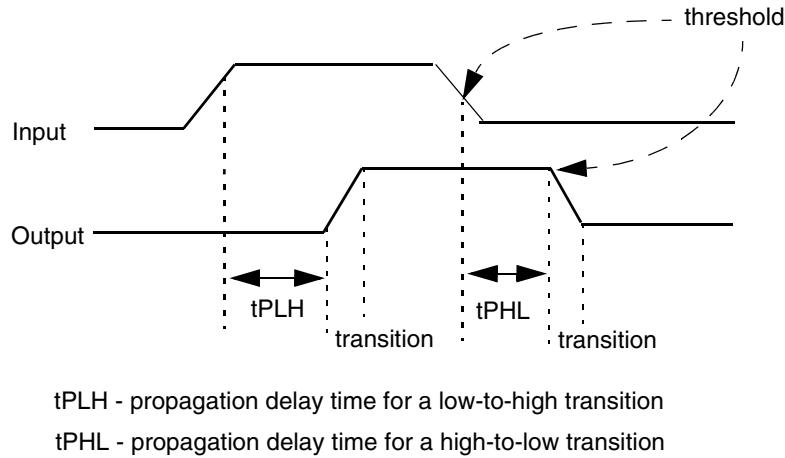
Example

```
cell (general) {  
    pin(a) {  
        direction: output;  
        timing() {  
            cell_rise(basic_template) {  
                values ("0.0, 0.13, 0.17, 0.19", "0.21, 0.23, \\\n                    0.30, 0.41", "0.22, 0.31, 0.35, 0.47",\\\n                    "0.33, 0.37, 0.45, 0.50");  
            }  
        }  
    }  
}
```

`rise_propagation` and `fall_propagation` Groups

The propagation delay is the delay between the thresholds in the transition at the input of a gate and the start of the output transition, as shown in [Figure 11-5](#).

If you define propagation delay tables for a timing arc, you cannot specify cell delay tables for that arc.

Figure 11-5 Propagation Delay Time Waveforms

tPLH - propagation delay time for a low-to-high transition

tPHL - propagation delay time for a high-to-low transition

Example

```
pin(Z) {
    ...
    timing() {
        ...
        rise_propagation(scalar) {
            values ("0.12");
        }
        fall_propagation(scalar) {
            values ("0.12");
        }
        rise_transition(tran_template) {
            values ("0.1, 0.15, 0.20, 0.29");
        }
        fall_transition (tran_template) {
            values ("0.1, 0.15, 0.20, 0.29");
        }
        related_pin : "A B" ;
    }
}
```

See “[Specifying Delay Scaling Attributes](#)” on page 6-17 for information about calculating delay factors.

[Example 11-4](#) shows the use of lookup tables and templates for describing cell delay.

Example 11-4 Using Templates in NLDML

```
library( vendor_a ) {
    /* Use CMOS nonlinear delay model */
    delay_model : table_lookup;
    ...
    /* Define template of size 4 x 4*/
```

```

lu_table_template(basic_template) {
    variable_1 : input_net_transition;
    variable_2 : total_output_net_capacitance;
    index_1 ("0.0, 0.5, 1.5, 2.0");
    index_2 ("0.0, 2.0, 4.0, 6.0");
}
/* Define library-level one-dimensional lu_table of size 4 */
lu_table_template(one_dimensional) {
    variable_1 : input_net_transition;
    index_1 ("0.0, 0.5, 1.5, 2.0");
}
. . .
/* Define a cell with pins containing lu_table groups that */
/* inherit the library-level template information */
cell (general) {
    . . .
    pin(a) {
        direction: output;
        timing() {
            . . .
            /* Inherit the 'basic_template' template */
            cell_rise(basic_template) {
                /* Specify all the values */
                values ("0.0, 0.13, 0.17, 0.19", "0.21, 0.23, 0.30, \
                        0.41", "0.22, 0.31, 0.35, 0.47", "0.33, \
                        0.37, 0.45, 0.50");
            }
        }
    }
    . . .
}
pin(b) {
    direction: output;
    timing() {
        . . .
        /* Inherit the 'one-dimensional' template */
        cell_rise(one_dimensional) {
            /*Specify all the values within a pair of ""*/
            values ("0.1, 0.15, 0.20, 0.29");
        }
    }
}
. . .
pin(c) {
    direction: output;
    timing() {
        . . .
        /* Use the predefined 'scalar' template */
        cell_rise(scalar) {
            /* Specify the value */
            values ("0.12");
        }
    }
}
. . .
}
. . .
*/
/* The rest of the library. */

```

```
} . . .
```

retaining_rise and retaining_fall Groups

The retaining delay is the time during which an output port retains its current logical value after a voltage rise or fall at a related input port.

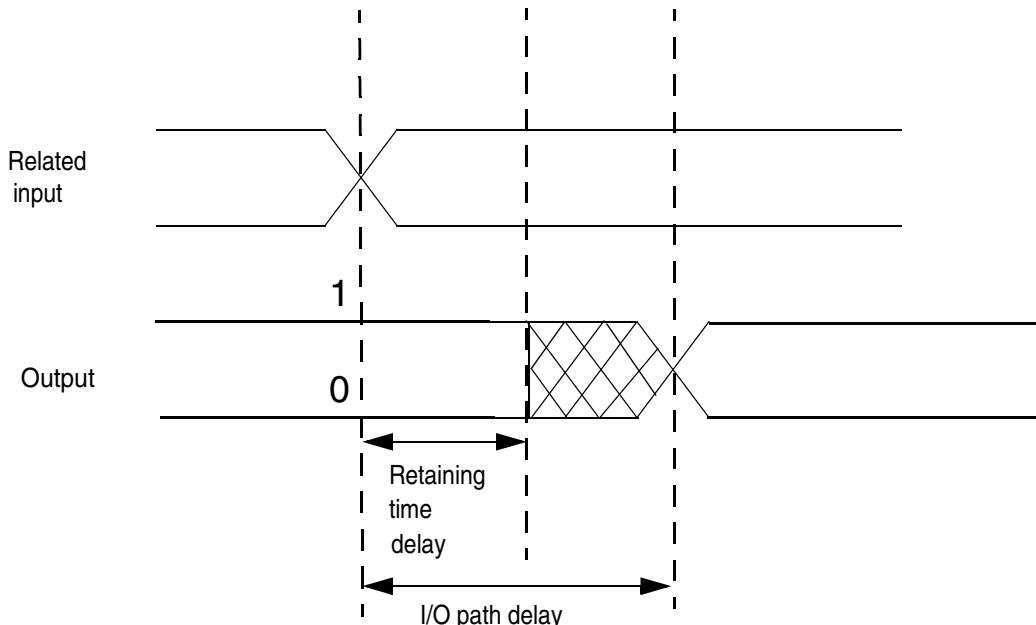
The retaining delay is part of the arc delay (I/O path delay); therefore, its time cannot exceed the arc delay time. Because retaining delay is part of the arc delay, the retaining delay tables are placed within the timing arc.

The value you enter for the `retaining_rise` attribute determines how long the output pin retains its current value, 0, after the value at the related input port has changed.

The value you enter for the `retaining_fall` attribute determines how long the output retains its current value, 1, after the value at the related input port has changed.

[Figure 11-6](#) shows retaining delay in regard to changes in a related input port.

Figure 11-6 Retaining Time Delay



[Example 11-5](#) shows how to use the `retaining_rise` and `retaining_fall` attributes.

Example 11-5 Retaining Time Delay

```
library(foo) {
...
lu_table_template (retaining_table_template){
...
variable_1: total_output_net_capacitance;
```

```

variable_2: input_net_transition;
index_1 ("0.0, 1.5");
index_2 ("1.0, 2.1");
}
...
cell (cell_name){
...
pin (A) {
  direction : output;
...
  timing(){
    related_pin : "B"
...
    retaining_rise (retaining_table_template){
      values ("0.00, 0.23", "0.11, 0.28");
    }
    retaining_fall (retaining_table_template){
      values ("0.01, 0.30", 0.12, 0.18");
    }
  }/*end of pin() */
...
}/*end of cell() */
...
}/*end of library() */

```

See “[Specifying Delay Scaling Attributes](#)” on page 6-17 for information about calculating delay factors.

retain_rise_slew and retain_fall_slew Groups

These groups let you specify a slew table for the retain arc that is separate from the table of the parent delay arc. This retain arc represents the time it takes until an output pin starts losing its current logical value after a related input pin is changed. This decaying of the output logic value occurs at a different time than the propagation of the final logical value, and at a different rate.

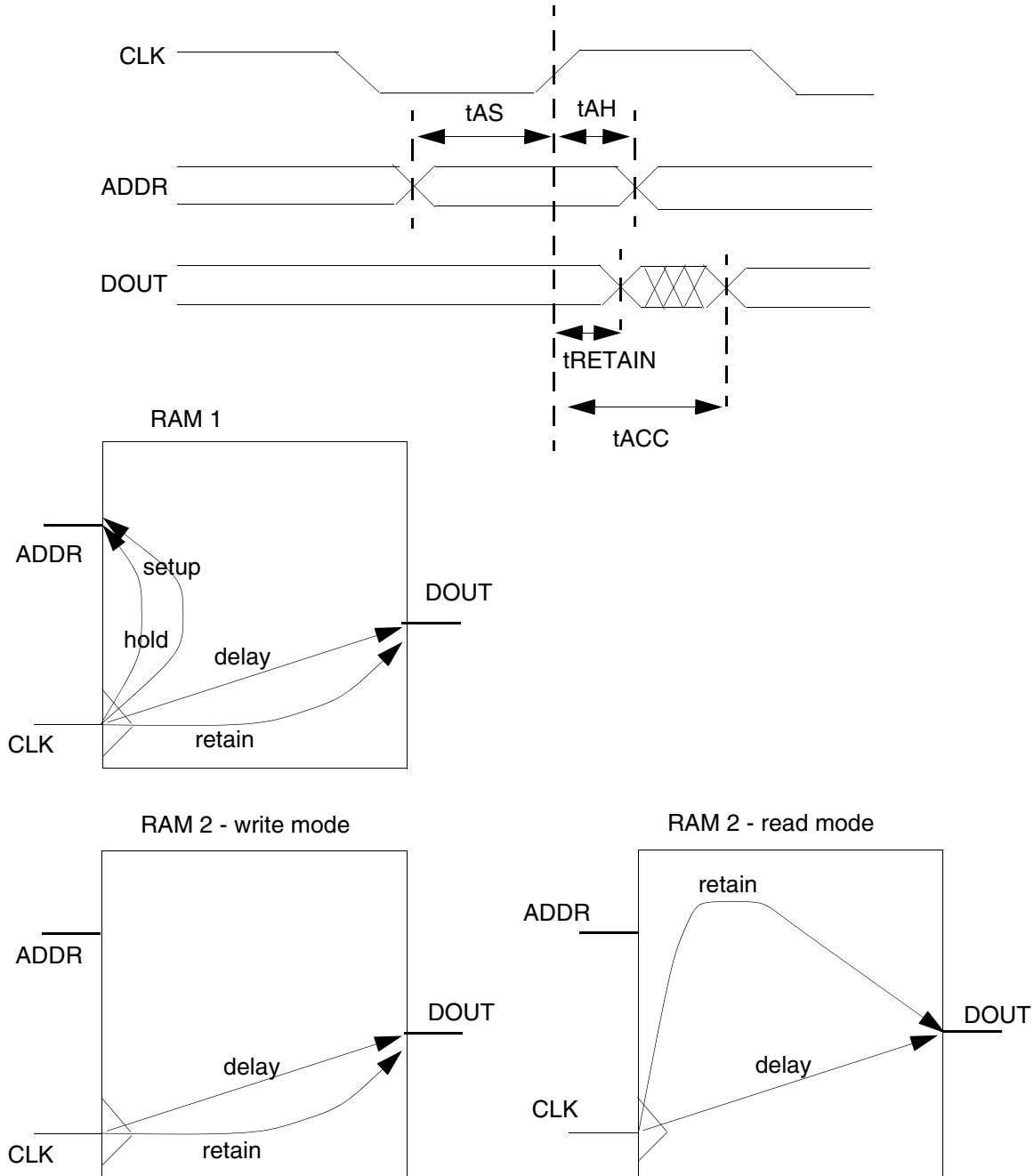
The retain delay is part of the arc delay (I/O path delay), and therefore its time cannot exceed the arc delay time. Because the retain delay is part of the arc delay, the retain delay tables are placed within the timing arc.

The value you enter for the `retain_rise_slew` attribute determines how long the output pin retains its current value, 0, after the value at the related input port has changed.

The value you enter for the `retain_fall_slew` attribute determines how long the output retains its current value, 1, after the value at the related input port has changed.

[Figure 11-7](#) shows a timing diagram of synchronous RAM.

Figure 11-7 Timing Diagram of Synchronous RAM



Example

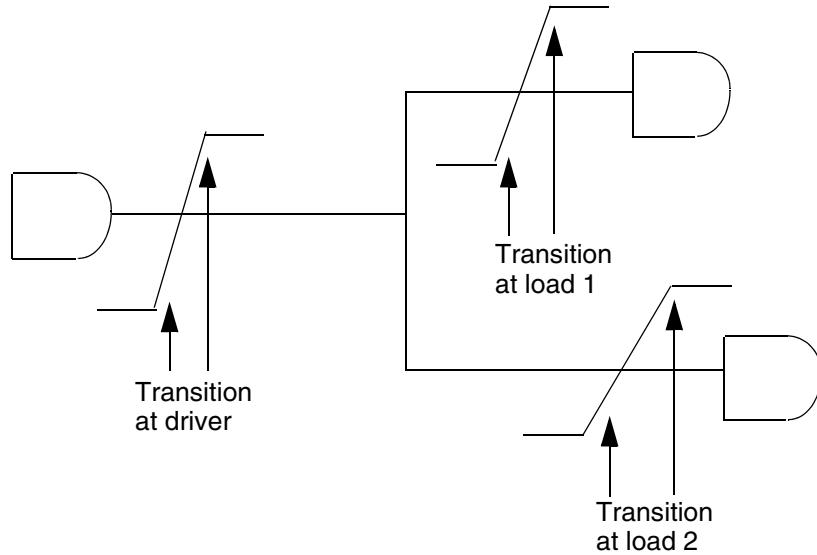
```
library(library_name) {
    ...
    lu_table_template (retaining_table_template){
        ...
        variable_1: total_output_net_capacitance;
        variable_2: input_net_transition;
        index_1 ("0.0, 1.5");
        index_2 ("1.0, 2.1");
    }
    ...
    cell (cell_name){
        ...
        pin (A) {
            direction : output;
            ...
            timing(){
                related_pin : "B"
                ...
                retaining_rise (retaining_table_template){
                    values ("0.00, 0.23", "0.11, 0.28");
                }
                retaining_fall (retaining_table_template){
                    values ("0.01, 0.30", 0.12, 0.18");
                }
                retain_rise_slew(retaining_time_template){
                    values("0.01,0.02");
                }
                retain_fall_slew(retaining_time_template){
                    values("0.01,0.02");
                }
            }/*end of pin() */
        ...
    }/*end of cell() */
    ...
}/*end of library() */
```

See “[Specifying Delay Scaling Attributes](#)” on page [6-17](#) for information about calculating delay factors.

Modeling Transition Time Degradation

Current nonlinear delay models are based on the assumption that the transition time at the load pin is the same as the transition time created at the driver pin. In reality, the net acts as a low-pass filter and the transition flattens out as it propagates from the driver of the net to each load, as shown in [Figure 11-8](#). The higher the interconnect load, the greater the flattening effect and the greater the transition delay.

Figure 11-8 Transition Time Degradation



To model the degradation of the transition time as it propagates from an output pin over the net to the next input pin, include these library-level groups in your library:

- `rise_transition_degradation`
- `fall_transition_degradation`

These groups contain the values describing the degradation functions for rise and fall transitions in the form of lookup tables. The lookup tables are indexed by

- Transition time at the net driver
- Connect delay between the driver and a load

These are the supported values for transition degradation (`variable_1` and `variable_2`):

- `output_pin_transition`
- `connect_delay`

You can assign either `connect_delay` or `output_pin_transition` to `variable_1` or `variable_2`, if the index and table values are consistent with the assignment.

The values you use in the table compute the degradation transition according to the following formula:

```
degraded_transition =
table_lookup(f(output_pin_transition, connect_delay))
```

Design Compiler uses transition degradation tables when it indexes into any delay table in a library that uses the table parameters `input_net_transition`, `constrained_pin_transition`, or `related_pin_transition` in an `lu_table_template` group. Transition degradation tables in a library have no effect on computations related to cells from other libraries.

The k-factors for process, voltage, and temperature are not supplied for the new tables. The `output_pin_transition` value and the `connect_delay` value are computed at the current, rather than nominal, operating conditions.

[Example 11-6](#) shows the use of the degradation tables. In this example, `trans_deg` is the name of the template for the transition degradation.

Example 11-6 Using Degradation Tables

```
library (simple_tlu) {
delay_model : table_lookup;

/* define the table templates */

lu_table_template(prop) {
    variable_1 : input_net_transition ;
    variable_2 : total_output_net_capacitance ;
    index_1("0, 1, 2");
    index_2("0, 1, 2");
}

lu_table_template(tran) {
    variable_1 : total_output_net_capacitance ;
    variable_2 : input_net_transition ;
    index_1("0, 1, 2");
    index_2("0, 1, 2");
}

lu_table_template(constraint) {
    variable_1 : constrained_pin_transition ;
    index_1("0, 1, 2");
    variable_2 : related_pin_transition ;
    index_2("0, 1, 2");
}

lu_table_template(trans_deg) {
    variable_1 : output_pin_transition ;
    index_1("0, 1");
    variable_2 : connect_delay ;
    index_2("0, 1");
}

/* the new degradation tables */

rise_transition_degradation(trans_deg) {
    values("0.0, 0.6", "1.0, 1.6");
}
```

```

}
fall_transition_degradation(trans_deg) {
    values("0.0, 0.8", "1.0, 1.8");
}

/* other library level defaults */

default inout pin_cap : 1.0;
...
k_process_fall_transition : 1.0;
...

nom_process : 1.0;
nom_temperature : 25.0;
nom_voltage : 5.0;

operating_conditions(BASIC_WORST) {
    process : 1.5 ;
    temperature : 70 ;
    voltage : 4.75 ;
    tree_type : "worst_case_tree" ;
}

/* list of cell descriptions */
cell(AN2) {
.....
}

```

Any linear function of `output_pin_transition` and `connect_delay` can be represented with four table points, because the Design Compiler interpolation and extrapolation mechanism is linear. Larger tables are required to represent more complex degradation functions, and breakpoints must be chosen so that the interpolation error is acceptable.

Modeling Load Dependency

[“Describing Transition Delay” on page 11-40](#) describes how to model the transition time dependency of a constrained pin and its related pin on timing constraints. You can further model the effect of unbuffered output on timing constraints by modeling load dependency. Constraints can also be load-dependent.

This is the procedure for modeling load dependency.

1. In the `timing` group of the output pin, set the `timing_type` attribute to one of the values shown in [Table 11-4](#).
2. Use the `related_output_pin` attribute in the `timing` group to specify which output pin to use to calculate the load dependency.

3. Create a three-dimensional table template that uses two variables and indexes to model transition time and the third variable and index to model load. The variable values for representing output loading on the `related_output_pin` are

```
related_out_total_output_net_capacitance
related_out_output_net_length
related_out_output_net_wire_cap
related_out_output_net_pin_cap
```

See “[Defining the NLDM Template](#)” on page 11-14.

4. Create a three-dimensional lookup table, using the table template and the `index_3` attribute in the lookup table group. (See “[Creating Lookup Tables](#)” on page 11-18.) The following groups are valid lookup tables for output load modeling:

- `rise_constraint`
- `fall_constraint`

See “[Setting Setup and Hold Constraints](#)” on page 11-57 for information about these groups.

[Example 11-7](#) is an example of a library that includes a load-dependent model.

Example 11-7 Load-Dependent Model in a Library

```
library(load_dependent) {
    delay_model : table_lookup;
    ...
    lu_table_template(constraint) {
        variable_1 : constrained_pin_transition;
        variable_2 : related_pin_transition;
        variable_3 : related_out_total_output_net_capacitance;
        index_1 ("1, 5, 10");
        index_2 ("1, 5, 10");
        index_3 ("1, 5, 10");
    }
    cell(selector) {
        ...
        pin(d) {
            direction : input ;
            capacitance : 4 ;
            timing() {
                related_pin : "sel";
                related_output_pin : "so";
                timing_type : non_seq_hold_rising;
                rise_constraint(constraint) {
                    values("1.5, 2.5, 3.5", "1.6, 2.6, 3.6", "1.7, 2.7, 3.7", \
                        "1.8, 2.8, 3.8", "1.9, 2.9, 3.9", "2.0, 3.0, 4.0", \
                        "2.1, 3.1, 4.1", "2.2, 3.2, 4.2", "2.3, 3.3, 4.3");
                }
                fall_constraint(constraint) {
                    values("1.5, 2.5, 3.5", "1.6, 2.6, 3.6", "1.7, 2.7, 3.7", \
                        "1.8, 2.8, 3.8", "1.9, 2.9, 3.9", "2.0, 3.0, 4.0", \
                        "2.1, 3.1, 4.1", "2.2, 3.2, 4.2", "2.3, 3.3, 4.3");
                }
            }
        }
    }
}
```

```
    }
}
...
}
...
}
```

Describing Slope Sensitivity

The slope delay of an element is the incremental time delay due to slowing changing input signals. Slope delay is calculated with the transition delay at the previous output pin with a slope sensitivity factor.

A slope sensitivity factor accounts for the time during which the input voltage begins to rise but has not reached the threshold level at which channel conduction begins.

Describing State-Dependent Delays

These timing attributes describe the delay values for specified conditions.

In the `timing` group of a technology library, you need to specify state-dependent delays that correspond to entries in Open Verilog International Standard Delay Format (OVI SDF 2.1) syntax. See the *Synopsys Logic Library Reference Manual* for syntax information about the attributes used to define timing check conditions and edge information.

To define a state-dependent timing arc, use these attributes:

- `when`
- `sdf_cond`

For state-dependent timing, each `timing` group requires both the `sdf_cond` and the `when` attributes.

You must define mutually exclusive conditions for state-dependent timing arcs. *Mutually exclusive* means that no more than one condition (defined in the `when` attribute) can be met at any time. Use the `default_timing` attribute to specify a default timing arc in the case of multiple timing arcs with `when` attributes.

See “[Generating an SDF File](#)” on page 11-79 for information about conditional path delays in an SDF file.

when Simple Attribute

The `when` attribute is a Boolean expression in the `timing` group that specifies the condition on which a timing arc depends to activate a path. Conditional timing lets you control the output pin of a cell with respect to the various *states* of the input pins.

See [Table 11-5](#) for the valid Boolean operators.

Table 11-5 Valid Boolean Operators

Operator	Description
,	Invert previous expression
!	Invert following expression
^	Logical XOR
*	Logical AND
&	Logical AND
space	Logical AND
+	Logical OR
	Logical OR
1	Signal tied to logic 1
0	Signal tied to logic 0

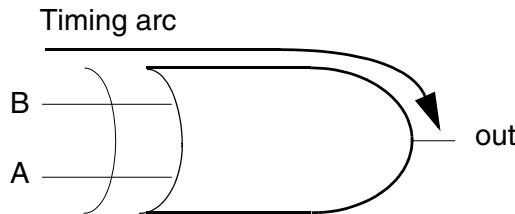
The order of precedence of the operators is left to right, with inversion performed first, then XOR, then AND, then OR.

Example

```
when : "B";
```

[Figure 11-9](#) shows an XOR gate.

Figure 11-9 XOR Gate With State-Dependent Timing Arc



[Example 11-8](#) shows how to use the `when` attribute for an XOR gate. In the description of the XOR cell, pin A sets conditional timing for the output pin “out” when you define the timing arc for `related_pin` B. In this example, when you set conditional timing for `related_pin` A with the `when : "B"` statement, the output pin gets the `negative_unate` value of A when the condition is B.

There are limitations on the pin types that can be used with different types of cells with the `when` attribute.

For a combinational cell, these pins are valid with the `when` attribute:

- Pins in the `function` attribute for regular combinational timing arcs
- Pins in the `three_state` attribute for the endpoint of the timing arc

For a sequential cell, valid pins or variables to use with the `when` attribute are determined by the timing type of the arc.

- For timing types `rising_edge` and `falling_edge`: Pins in these attributes are allowed in the `when` attribute:
 - `next_state`
 - `clocked_on`
 - `clocked_on_also`
 - `enable`
 - `data_in`
- For timing type `clear`
 - If the pin’s function is the first state variable in the `flip-flop` or `latch` group, the pin that defines the clear condition in the `flip-flop` or `latch` group is allowed in the `when` construct.
 - If the pin’s function is the second state variable in the `flip-flop` or `latch` group, the pin that defines the preset condition in the `flip-flop` or `latch` group is allowed in the `when` construct.

- For timing type `preset`
 - If the pin's function is the first state variable in the `flip-flop or latch` group, the pin that defines the preset condition in the `flip-flop or latch` group is allowed in the `when` construct.
 - If the pin's function is the second state variable in the `flip-flop or latch` group, the pin that defines the clear condition in the `flip-flop or latch` group is allowed in the `when` construct.

See “[timing_type Simple Attribute](#)” on page 11-22 for more information.

All input pins in a black box cell (a cell without a `function` attribute) are allowed in the `when` attribute.

sdf_cond Simple Attribute

Defined in the state-dependent timing group, the `sdf_cond` attribute supports Standard Delay Format (SDF) file generation and condition matching during back-annotation.

Example

```
sdf_cond : "SE ==1'B1";
```

The `sdf_cond` attribute must be logically equivalent to the `when` attribute for the same timing arc. If the two Boolean expressions are not equivalent, back-annotation is not performed properly.

The Library Compiler tool does not parse the `sdf_cond` expression, nor does it check for logical equivalence between the `sdf_cond` and `when` Boolean expressions. The `sdf_cond` expressions must be syntax-compliant with SDF 2.1. If the expressions do not meet this standard, errors are generated later in the flow during the generation and reuse of the SDF files.

For simple delay paths, such as IOPATH, you can use the Boolean operators, such as `&&` and `||`, with the `sdf_cond` attribute. However, Verilog timing check statements, including setup, hold, recovery, and removal do not support Boolean operators.

Synopsys tools, such as PrimeTime, Design Compiler, and IC Compiler use the `when` expression instead of the `sdf_cond` expression. For a library file with the `sdf_cond` attribute, these tools:

- Read the corresponding SDF file without issuing an error.
- Write the `sdf_cond` expressions in the SDF file.

[Example 11-8](#) is a 2-input XOR gate. It represents a commonly used state-dependent delay case. The delay between pin A and pin OUT is 1.3 for rising and 1.5 for falling when pin B =

1. There is an additional timing arc between the same two pins that has a rise delay of 1.4 and a fall delay of 1.6 when pin B = 0.

Example 11-8 XOR Cell With State-Dependent Timing

```
cell(XOR) {
    pin(A) {
        direction : input;
        ...
    }
    pin(B) {
        direction : input;
        ...
    }
    pin(out) {
        direction : output;
        function : "A ^ B";
        timing() {
            related_pin : "A";
            timing_sense : negative_unate;
            when : "B";
            sdf_cond : " B == 1'B1 ";
            cell_rise(scalar) {
                values( " 1.3 " );
            }
            cell_fall(scalar) {
                values( " 1.5 " );
            }
        }
        timing() {
            related_pin : "A";
            timing_sense : positive_unate;
            when : "!B";
            sdf_cond : " B == 1'B0 ";
            cell_rise(scalar) {
                values( " 1.4 " );
            }
            cell_fall(scalar) {
                values( " 1.6 " );
            }
        }
        timing() /* default timing arc */
        related_pin : "A";
        timing_sense : non_unate;
        cell_rise(scalar) {
            values( " 1.4 " );
        }
        cell_fall(scalar) {
            values( " 1.6 " );
        }
    }
    timing() {
        related_pin : "B";
        timing_sense : negative_unate;
```

```

        when : "A";
        sdf_cond : "A == 1'B1 ";
        cell_rise(scalar) {
            values( " 1.3 ");
        }
        cell_fall(scalar) {
            values( " 1.5 ");
        }
    }
timing() {
    related_pin : "B";
    timing_sense : positive_unate;
    when : "!A";
    sdf_cond : "A == 1'B0 ";
    cell_rise(scalar) {
        values( " 1.4 ");
    }
    cell_fall(scalar) {
        values( " 1.6 ");
    }
}
timing() /* default timing arc */
{
    related_pin : "B";
    timing_sense : non_unate;
    cell_rise(scalar) {
        values( " 1.4 ");
    }
    cell_fall(scalar) {
        values( " 1.6 ");
    }
}
}
}

```

Setting Setup and Hold Constraints

Signals arriving at an input pin have ramp times. Therefore, you must ensure that the data signal has stabilized before latching its value by defining setup and hold arcs as timing requirements. No path tracing takes place along these timing arcs; they are used solely for constraint verification.

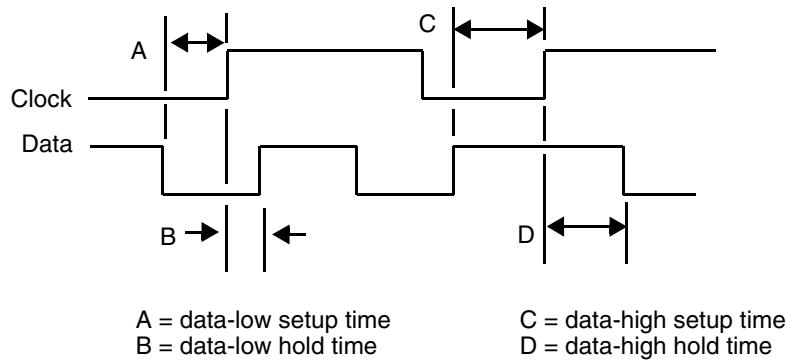
- Setup constraints describe the minimum time allowed between the arrival of the data and the transition of the clock signal. During this time, the data signal must remain constant. If the data signal makes a transition during the setup time, an incorrect value might be latched.
 - Hold constraints describe the minimum time allowed between the transition of the clock signal and the latching of the data. During this time, the data signal must remain

constant. If the data signal makes a transition during the hold time, an incorrect value might be latched.

By combining a setup time and a hold time, you can ensure the stability of data that is latched.

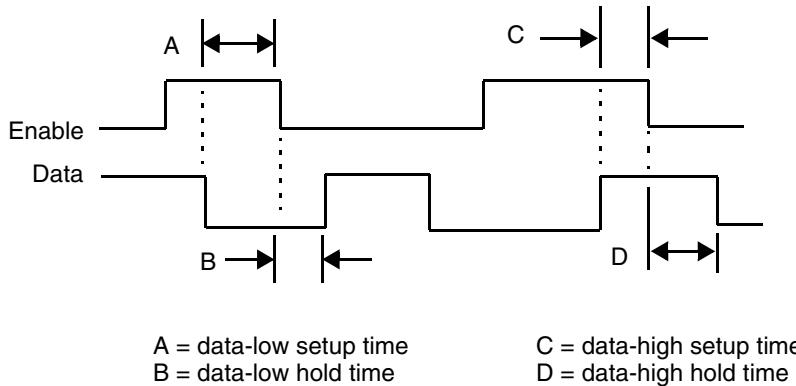
[Figure 11-10](#) shows setup and hold timing for a rising-edge-triggered flip-flop. The timing checks for flip-flops use the activating edge of the clock, which is the rising edge in this case.

Figure 11-10 Setup and Hold Constraints for Rising-Edge-Triggered Flip-Flop



[Figure 11-11](#) illustrates setup and hold timing for a high-enable latch. The timing checks for latches generally use the deactivating edge of the enable signal, which is the falling edge in this case. However, the method used depends on the vendor.

Figure 11-11 Setup and Hold Constraints for High-Enable Latch



NLDM supports timing constraints sensitive to clock or data-input transition times.

Each constraint is defined by a `timing` group with two lookup tables:

- `rise_constraint group`
- `fall_constraint group`

rise_constraint and fall_constraint Groups

These are constraint tables. The format of the lookup table template and the format of the lookup table are the same as described previously in “[Defining the NLDM Template](#)” on page 11-14 and “[Creating Lookup Tables](#)” on page 11-18.

These are valid variable values for the timing constraint template:

`constrained_pin_transition`

Value for the transition time of the pin that owns the `timing` group.

`related_pin_transition`

Value for the transition time of the `related_pin` defined in the `timing` group.

For each `timing` group containing one of the following `timing_type` attribute values, at least one lookup table is required:

- `setup_rising`
- `setup_falling`
- `hold_rising`
- `hold_falling`
- `skew_rising`
- `skew_falling`
- `non_seq_setup_rising`
- `non_seq_setup_falling`
- `non_seq_hold_rising`
- `non_seq_hold_falling`
- `nochange_high_high`
- `nochange_high_low`
- `nochange_low_high`
- `nochange_low_low`

For each `timing` group with one of the following `timing_type` attribute values, only one lookup table is required:

- `recovery_rising`
- `recovery_falling`
- `removal_rising`
- `removal_falling`

[Example 11-9](#) shows how to use tables to specify setup constraints for a flip-flop.

Example 11-9 CMOS Nonlinear Timing Model Using Constraint

```
library( vendor_b ) {

/* 1. Use delay lookup table */
delay_model : table_lookup;

/* 2. Define template of size 3 x 3*/
lu_table_template(constraint_template) {
    variable_1 : constrained_pin_transition;
    variable_2 : related_pin_transition;
    index_1 ("0.0, 0.5, 1.5");
    index_2 ("0.0, 2.0, 4.0");
}

. . .

cell(dff) {
    pin(d) {
        direction: input;
        timing() {
            related_pin : "clk";
            timing_type : setup_rising;
            /* Inherit the constraint_template template */
            rise_constraint(constraint_template) {
                /* Specify all the values */
                values ("0.0, 0.13, 0.19",
                        "0.21, 0.23, 0.41",
                        "0.33, 0.37, 0.50");
            }
            fall_constraint(constraint_template) {
                values ("0.0, 0.14, 0.20",
                        "0.22, 0.24, 0.42",
                        "0.34, 0.38, 0.51");
            }
        }
    }
. . .
}
}
```

Identifying Interdependent Setup and Hold Constraints

To reduce slack violation, use pairs of `interdependence_id` attributes to identify interdependent pairs of setup and hold constraint tables. Interdependence data is supported in conditional constraint checking. The `interdependence_id` increases independently for each condition. Interdependence data can be specified in pin or bus and bundle groups. For details, see the *Synopsys Logic Library Reference Manual*.

Setting Nonsequential Timing Constraints

You can set constraints requiring that the data signal on an input pin remain stable for a specified amount of time before or after another pin in the same cell changes state. These cells are termed nonsequential cells, because the related pin is not a clock signal.

All Synopsys delay models supporting sequential setup and hold constraint modeling also support nonsequential setup and hold modeling.

Scaling of nonsequential setup and hold constraints based on the environment use k-factors for sequential setup and hold constraints.

The values you can assign to a `timing_type` attribute to model nonsequential setup and hold constraints are

`non_seq_setup_rising`

Designates the rising edge of the related pin for the setup check.

`non_seq_setup_falling`

Designates the falling edge of the related pin for the setup check.

`non_seq_hold_rising`

Designates the rising edge of the related pin for the hold check.

`non_seq_hold_falling`

Designates the falling edge of the related pin for the hold check.

To model nonsequential setup and hold constraints for a cell,

1. Assign a value to the `timing_type` attribute in a `timing` group of an input or I/O pin.
2. Specify a related pin with the `related_pin` attribute in the `timing` group. The related pin in a timing arc is the pin used for the timing check.

Use any pin in the same cell, except for output pins, and the constrained pin itself as the related pin.

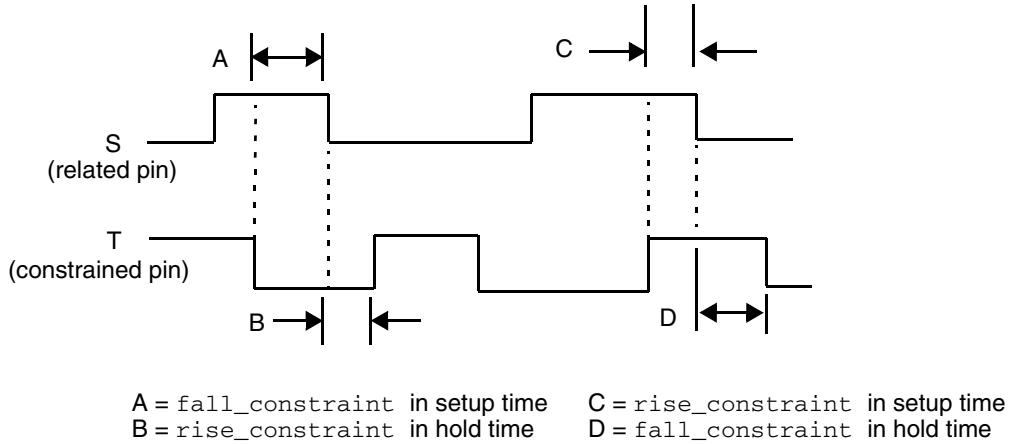
You can use both rising and falling edges as the active edge of the related pin for one cell.

Example

```
pin(T) {
    timing() {
        timing_type : non_seq_setup_falling;
        rise_constraint(scalar) {
            values (" 1.5 ");
        }
        fall_constraint(scalar) {
            values (" 1.5 ");
        }
        related_pin : "S";
    }
}
```

[Figure 11-12](#) shows the waveforms for the nonsequential timing arc described in the preceding example. In this timing arc, the constrained pin is T and its related pin is S. The intrinsic rise value describes setup time C or hold time B. The intrinsic fall value describes setup time A or hold time D.

Figure 11-12 Nonsequential Setup and Hold Constraints



Setting Recovery and Removal Timing Constraints

Use the recovery and removal timing arcs for asynchronous control pins such as clear and preset.

Recovery Constraints

The recovery timing arc describes the minimum allowable time between the control pin transition to the inactive state and the active edge of the synchronous clock signal (time between the control signal going inactive and the clock edge that latches data in).

The asynchronous control signal must remain constant during this time, or else an incorrect value might appear at the outputs.

[Figure 11-13](#) shows the recovery timing arc for a rising-edge-triggered flip-flop with active-low clear.

[Figure 11-14](#) shows the recovery timing arc for a low-enable latch with active-high preset.

Figure 11-13 Recovery Timing Constraint for a Rising-Edge-Triggered Flip-Flop

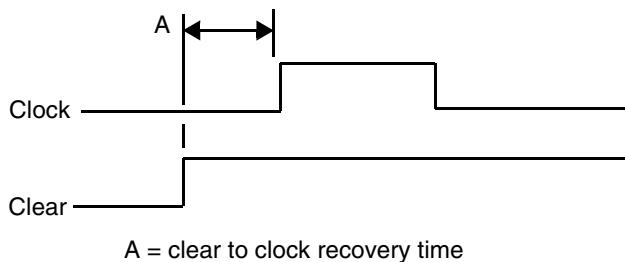
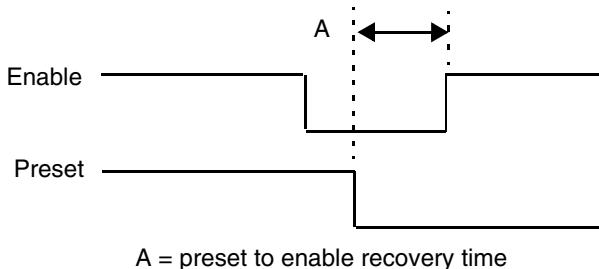


Figure 11-14 Recovery Timing Constraint for a Low-Enable Latch



The values you can assign to a `timing_type` attribute to define a recovery time constraint are

`recovery_rising`

Uses the rising edge of the related pin for the recovery time check; the clock is rising-edge-triggered.

```
recovery_falling
```

Uses the falling edge of the related pin for the recovery time check; the clock is falling-edge-triggered.

To define a recovery time constraint for an asynchronous control pin,

1. Assign a value to the `timing_type` attribute.

Use `recovery_rising` for rising-edge-triggered flip-flops and low-enable latches; use `recovery_falling` for negative-edge-triggered flip-flops and high-enable latches.

2. Identify the synchronous clock pin as the `related_pin`.

For active-low control signals, define the recovery time with the `rise_constraint` statement.

For active-high control signals, define the recovery time with the `fall_constraint` statement.

Example

This example shows a recovery timing arc for the active-low clear signal in a rising-edge-triggered flip-flop. The `rise_constraint` value represents clock recovery time A in [Figure 11-13](#).

```
pin (Clear) {
    direction : input ;
    capacitance : 1 ;
    timing() {
        related_pin : "Clock" ;
        timing_type : recovery_rising;
        rise_constraint(scalar) {
            values (" 1.0 " ) ;
        }
    }
}
```

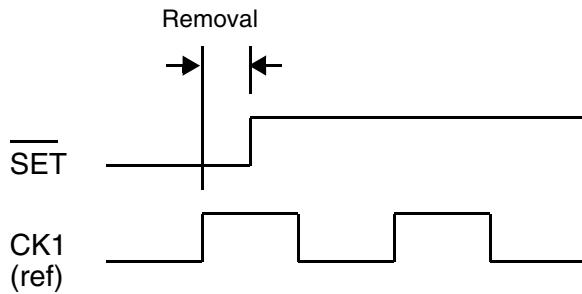
The following example shows a recovery timing arc for the active-high preset signal in a low-enable latch. The `fall_constraint` value represents clock recovery time A in [Figure 11-14](#).

```
pin (Preset) {
    direction : input ;
    capacitance : 1 ;
    timing() {
        related_pin : "Enable" ;
        timing_type : recovery_rising;
        fall_constraint(scalar) {
            values (" 1.0 " ) ;
        }
    }
}
```

Removal Constraint

This constraint is also known as the asynchronous control signal hold time. The removal constraint describes the minimum allowable time between the active edge of the clock pin while the asynchronous pin is active and the inactive edge of the same asynchronous control pin (see [Figure 11-15](#)).

Figure 11-15 Timing Diagram for Removal Constraint



The values you can assign to a `timing_type` attribute to define a removal constraint are `removal_rising`

Use when the cell is a low-enable latch or a rising-edge-triggered flip-flop.

`removal_falling`

Use when the cell is a high-enable latch or a falling-edge-triggered flip-flop.

To define a removal constraint,

1. Assign a value to the `timing_type` attribute.
2. Identify the synchronous clock pin as the `related_pin`.
3. For active-low asynchronous control signals, define the removal time with the `rise_constraint` attribute.

For active-high asynchronous control signals, define the removal time with the `fall_constraint` attribute.

Example

```
pin ( SET ) {
    ...
    timing() {
        timing_type : removal_rising;
        related_pin : " CK1 ";
        rise_constraint(scalar) {
            values (" 1.0 " );
        }
    }
}
```

```
    }  
}
```

Setting No-Change Timing Constraints

You can model no-change timing checks to use in static timing verification during synthesis.

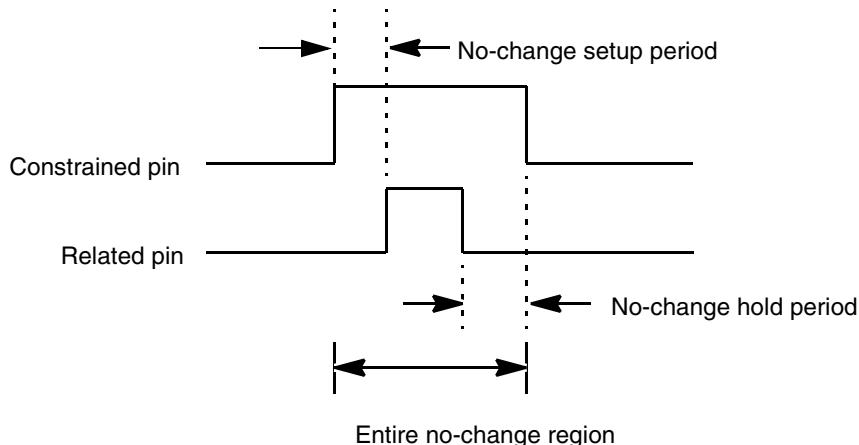
A no-change timing check checks a constrained signal against a level-sensitive related signal. The constrained signal must remain stable during an established setup period, for the width of the related pulse, and during an established hold period.

For example, you can use the no-change timing check to model the timing requirements of latch devices with latch enable signals. To ensure correct latch sampling, the latch enable signal must remain stable during the clock pulse and the setup and hold time around the clock pulse.

You can also use the no-change timing check to model the timing requirements of memory devices. To guarantee correct read/write operations, the address or data must remain stable during a read/write enable pulse and the setup and hold margins around the pulse.

[Figure 11-16](#) shows a no-change timing check between a constrained pin and its level-sensitive related pin.

Figure 11-16 No-Change Timing Check



The values you can assign to a `timing_type` attribute to define a no-change timing constraint are

`nochange_high_high`

Specifies a positive pulse on the constrained pin and a positive pulse on the related pin.

`nochange_high_low`

Specifies a positive pulse on the constrained pin and a negative pulse on the related pin.

`nochange_low_high`

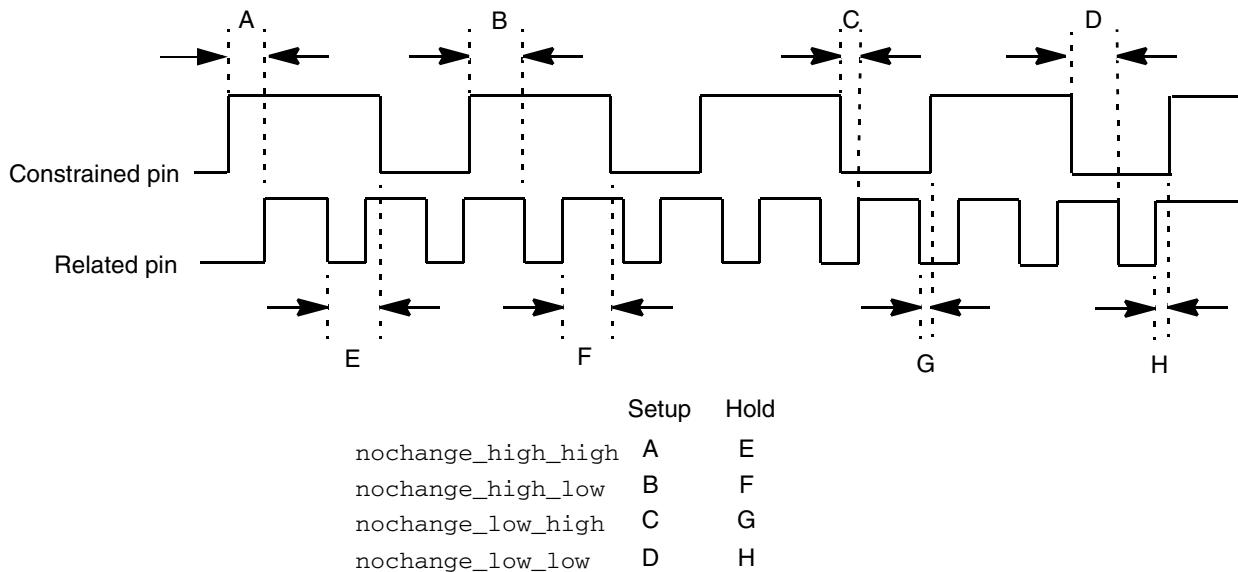
Specifies a negative pulse on the constrained pin and a positive pulse on the related pin.

`nochange_low_low`

Specifies a negative pulse on the constrained pin and a negative pulse on the related pin.

[Figure 11-17](#) shows the waveforms for these constraints.

Figure 11-17 No-Change Setup and Hold Constraint Waveforms



To model no-change timing constraints,

1. Assign a value to the `timing_type` attribute.
2. Specify a related pin with the `related_pin` attribute in the `timing` group.

The final high or low designation in the `timing_type` attribute tells Design Compiler which pulse of the related pin is active for the setup or hold constraint. The related pin in the timing arc is the pin used for the timing check.

3. Specify delay attribute values according to the delay model you use, as summarized in the following table.

No-change constraint	Setup attribute for nonlinear delay models	Hold attribute for nonlinear delay models
nochange_high_high	rise_constraint	fall_constraint
nochange_high_low	rise_constraint	fall_constraint
nochange_low_high	fall_constraint	rise_constraint
nochange_low_low	fall_constraint	rise_constraint

Note:

With no-change timing constraints, conditional timing constraints have different interpretations than they do with other constraints. See “[Setting Conditional Timing Constraints](#)” on page 11-72 for more information.

Specify setup time with the `rise_constraint` attribute and the hold time with the `fall_constraint` attribute.

This is the syntax for the no-change timing check:

```
timing () {
    timing_type : nochange_high_high | nochange_high_low | \
                  nochange_low_high | nochange_low_low ;
    related_pin : related_pinname;
    rise_constraint (template_name_id) {
        /* constrained signal rising */
        values (float, ..., float) ;
    }
    fall_constraint (template_name_id) {
        /* constrained signal falling */
        values (float, ..., float) ;
    }
}
```

Example

This is an example of a no-change timing check in a technology library:

```
library (the_lib) {
    delay_model : table_lookup;
    k_process_nochange_rise : 1.0; /* no-change scaling factors */
    k_process_nochange_fall : 1.0;
    k_volt_nochange_rise : 0.0;
    k_volt_nochange_fall : 0.0;
    k_temp_nochange_rise : 0.0;
    k_temp_nochange_fall : 0.0;
    cell (the_cell) {
```

```

pin(EN {
    timing () {
        timing_type : nochange_high_low;
        related_pin : CLK;
        rise_constraint (temp) { /* setup time */
            values (2.98);
        }
        fall_constraint (temp) { /* hold time */
            values (0.98);
        }
    }
    ...
}
...
...
}
...
}

```

In the CMOS Scalable Polynomial Delay Model

In the CMOS scalable polynomial delay model, specify setup time with `rise_constraint` and hold time with `fall_constraint`.

This is the syntax for the no-change timing check in the CMOS scalable polynomial delay model:

```

timing () {
    timing_type : nochange_high_high | nochange_high_low | \
                  nochange_low_high | nochange_low_low;
    related_pin : related_pinname;
    rise_constraint (poly_template_name_id) {
        /* constrained signal rising */
        orders(integer, integer) ;
        coefs(float, ..., float) ;
        variable_n_range(float, float) ;
    }
    fall_constraint (poly_template_name_id) {
        /* constrained signal falling */
        orders(integer, integer) ;
        coefs(float, ..., float) ;
        variable_n_range(float, float) ;
    }
}

```

Example

This is an example of a technology library no-change timing check using the CMOS scalable polynomial delay model:

```

library (the_lib) {
    delay_model : table_lookup;
    k_process_nochange_rise : 1.0; /* no-change scaling factors */
    k_process_nochange_fall : 1.0;
}

```

```

k_volt_nochange_rise : 0.0;
k_volt_nochange_fall : 0.0;
k_temp_nochange_rise : 0.0;
k_temp_nochange_fall : 0.0;
...
cell (the_cell) {
    pin(EN {
        timing () {
            timing_type : nochange_high_low;
            related_pin : CLK;
            rise_constraint (poly_template) { /* setup time */
                orders ("1, 1");
                coefs ("0.2487 +0.0520 -0.0268 -0.0053 " );
                variable_1_range (0.01, 3.00);
                variable_2_range (0.01, 3.00);
            }
            fall_constraint (poly_template) { /* hold time */
                orders ("1, 1");
                coefs ("0.2487 +0.0520 -0.0268 -0.0053 " );
                variable_1_range (0.01, 3.00);
                variable_2_range (0.01, 3.00);
            }
        }
        ...
    }
    ...
}
...
}

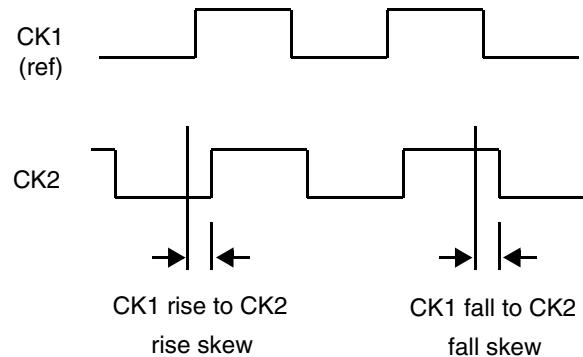
```

Setting Skew Constraints

The skew constraint defines the maximum separation time allowed between two clock signals.

Figure 11-18 is a timing diagram showing skew constraint.

Figure 11-18 Timing Diagram for Skew Constraint



The values you can assign to a `timing_type` attribute to define a skew constraint are `skew_rising`

The timing constraint interval is measured from the rising edge of the reference pin (specified in `related_pin`) to a transition edge of the parent pin of the `timing` group. The `rise_constraint` value is the maximum skew time between the reference pin rising and the parent pin rising. The `fall_constraint` value is the maximum skew time between the reference pin falling and the parent pin falling.

`skew_falling`

The timing constraint interval is measured from the falling edge of the reference pin (specified in `related_pin`) to a transition edge of the parent pin of the `timing` group. The `rise_constraint` value is the maximum skew time between the reference pin rising and the parent pin rising. The `fall_constraint` value is the maximum skew time between the reference pin falling and the parent pin falling.

To set skew constraint,

1. Assign a value to the `timing_type` attribute.
2. Define only one of these attributes in the `timing` group:
 - `rise_constraint`
 - `fall_constraint`
3. Use the `related_pin` attribute in the `timing` group to specify a reference clock pin. Only the following attributes in a skew timing group are used (all others are ignored):
 - `timing_type`
 - `related_pin`
 - `rise_constraint`
 - `fall_constraint`

Example

This example shows how to model constraint CK1 rise to CK2 rise skew:

```
pin (CK2) {
    ...
    timing() {
        timing_type : skew_rising;
        related_pin : " CK1 ";
        rise_constraint(scalar) { /* constrained signal rising */
            values (" float ");
        }
    }
}
```

Setting Conditional Timing Constraints

A conditional timing constraint describes a check Design Compiler performs when a specified condition is met. You can specify conditional timing checks in `pin`, `bus`, and `bundle` groups.

Use the following attributes and groups to specify conditional timing checks.

Attributes:

- `when`
- `sdf_cond`
- `when_start`
- `sdf_cond_start`
- `when_end`
- `sdf_cond_end`
- `sdf_edges`

Groups:

- `min_pulse_width`
- `minimum_period`

when and sdf_cond Simple Attributes

The `when` attribute defines enabling conditions for timing checks such as setup, hold, and recovery.

If you define `when`, you must define `sdf_cond`.

Using the `when` and `sdf_cond` pair is a short way of specifying `when_start`, `sdf_cond_start`, `when_end`, and `sdf_cond_end` when the start condition is identical to the end condition. The Library Compiler tool applies `when` and `sdf_cond` attribute values to both start and end conditions. The Library Compiler tool ignores the values for `when` and `sdf_cond` if the same timing group contains definitions for `when_start`, `sdf_cond_start`, `when_end`, and `sdf_cond_end`.

[“Describing State-Dependent Delays” on page 11-52](#) describes the `sdf_cond` and `when` attributes in defining state-dependent timing arcs.

when_start Simple Attribute

In a timing group, when_start defines a timing check condition specific to a start event in VHDL initiative toward ASIC libraries (VITAL) models in Synopsys logic format. The expression in this attribute contains any pin, including input, output, I/O, and internal pins. You must use real pin names. Bus and bundle names are not allowed.

Example

```
when_start : "SIG_A"; /*SIG_A must be a declared pin */
```

The when_start attribute requires an sdf_cond_start attribute in the same timing group.

The end condition is considered always true if a timing group contains when_start but no when_end.

sdf_cond_start Simple Attribute

In a timing group, sdf_cond_start defines a timing check condition specific to a start event in VITAL models in OVI SDF 2.1 syntax.

Example

```
sdf_cond_start : "SIG_A";
```

The sdf_cond_start attribute requires a when_start attribute in the same timing group.

The end condition is considered always true if a timing group contains sdf_cond_start but no sdf_cond_end.

when_end Simple Attribute

In a timing group, when_end defines a timing check condition specific to an end event in VITAL models in Synopsys logic format. The expression in this attribute contains any pin, including input, output, I/O, and internal pins. Pins must use real pin names. Bus and bundle names are not allowed.

Example

```
when_end : "CD * SD";
```

The when_end attribute requires an sdf_cond_end attribute in the same timing group.

The start condition is considered always true if a timing group contains when_end but no when_start.

sdf_cond_end Simple Attribute

In a timing group, `sdf_cond_end` defines a timing check condition specific to an end event in VITAL models in OVI SDF 2.1 syntax.

Example

```
sdf_cond_end : "SIG_0 == 1'b1";
```

The `sdf_cond_end` attribute requires a `when_end` attribute in the same timing group.

The start condition is considered always true if a timing group contains `sdf_cond_end` but no `sdf_cond_start`.

sdf_edges Simple Attribute

The `sdf_edges` attribute defines edge-specific information for both the start pins and the end pins in VITAL models. Edge types can be `noedge`, `start_edge`, `end_edge`, or `both_edges`. The default is `noedge`.

Example

```
sdf_edges : both_edges;
```

min_pulse_width Group

In a pin, bus, or bundle group, the `min_pulse_width` group models the enabling conditional minimum pulse width check. In the case of a pin, the timing check is performed on the pin itself, so the related pin must be the same.

The attributes in this group are `constraint_high`, `constraint_low`, `when`, and `sdf_cond`.

If the pin group contains a `min_pulse_width` group and either a `min_pulse_width_high` or `min_pulse_width_low` attribute, the Library Compiler tool ignores the attribute.

min_pulse_width Example

The following example shows the `min_pulse_width` group with the `constraint_high` and `constraint_low` attributes specified.

Example 11-10 min_pulse_width Example

```
min_pulse_width() {
    constraint_high : 3.0; /* min_pulse_width_high */
    constraint_low : 3.5; /* min_pulse_width_low */
    when : "SE";
    sdf_cond : "SE == 1'B1";
```

```
}
```

constraint_high and constraint_low Simple Attributes

At least one of these attributes must be defined in the `min_pulse_width` group. The `constraint_high` attribute defines the minimum length of time the pin must remain at logic 1. The `constraint_low` attribute defines the minimum length of time the pin must remain at logic 0.

when and sdf_cond Simple Attributes

These attributes define the enabling condition for the timing check: `when` in Synopsys logic format and `sdf_cond` in OVI SDF 2.1 syntax. Both attributes are required in the `min_pulse_width` group. All pins of the `cell`, `input`, `output`, `inout`, or `internal` group can be used with the `when` attribute.

minimum_period Group

In a `pin`, `bus`, or `bundle` group, the `minimum_period` group models the enabling conditional minimum period check. In the case of a pin, the check is performed on the pin itself, so the related pin must be the same. The attributes in this group are `constraint`, `when`, and `sdf_cond`.

If the `pin` group contains a `minimum_period` group and a `min_period` attribute, the Library Compiler tool ignores the `min_period` attribute.

minimum_period Example

```
minimum_period() {
    constraint : 9.5; /* min_period */
    when : "SE";
    sdf_cond : "SE == 1'B1";
}
```

constraint Simple Attribute

This required attribute defines the minimum clock period for the pin.

when and sdf_cond Simple Attributes

These attributes define the enabling condition for the timing check: `when` in Synopsys logic format and `sdf_cond` in OVI SDF 2.1 syntax. Both attributes are required in the `minimum_period` group. All pins of the `cell`, `input`, `output`, `inout`, or `internal` group can be used with the `when` attribute.

min_pulse_width and minimum_period Example

[Example 11-11](#) shows how to specify a lookup table with the `timing_type` attribute and `min_pulse_width` and `minimum_period` values. The `rise_constraint` group defines the rising pulse width constraint for `min_pulse_width`, and the `fall_constraint` group defines the falling pulse width constraint. For `minimum_period`, the `rise_constraint` group is used to model the period when the pulse is rising and the `fall_constraint` group is used to model the period when the pulse is falling. You can specify the `rise_constraint` group, the `fall_constraint` group, or both groups.

Example 11-11 A Library With timing_type Statements

```
library(example) {

    technology (cmos) ;
    delay_model : table_lookup ;

    /* 2-D table template */
    lu_table_template ( mpw ) {
        variable_1 : constrained_pin_transition;
        /* You can replace the constrained_pin_transition value with
           related_pin_transition, but you cannot specify both values. */
        variable_2 : related_out_total_output_net_capacitance;
        index_1("1, 2, 3");
        index_2("1, 2, 3");
    }

    /* 1-D table template */
    lu_table_template( f_ocap ) {
        variable_1 : total_output_net_capacitance;
        index_1 (" 0.0000, 1.0000 ");
    }

    cell( test ) {
        area : 200.000000 ;
        dont_use : true ;
        dont_touch : true ;

        pin ( CK ) {
            direction : input;
            rise_capacitance : 0.00146468;
            fall_capacitance : 0.00145175;
            capacitance : 0.00146468;
            clock : true;

            timing ( mpw_constraint) {
                related_pin : "CK";
                timing_type : min_pulse_width;
                related_output_pin : "Z";

                fall_constraint ( mpw) {

```

```

        index_1("0.1, 0.2, 0.3");
        index_2("0.1, 0.2");
        values( "0.10 0.11", \
            "0.12 0.13" \
            "0.14 0.15");
    }

    rise_constraint ( mpw) {
        index_1("0.1, 0.2, 0.3");
        index_2("0.1, 0.2");
        values( "0.10 0.11", \
            "0.12 0.13" \
            "0.14 0.15");

        timing ( mpw_constraint) {
            related_pin : "CK";
            timing_type : minimum_period;
            related_output_pin : "Z";

            fall_constraint ( mpw) {
                index_1("0.2, 0.4, 0.6");
                index_2("0.2, 0.4");
                values( "0.20 0.22", \
                    "0.24 0.26" \
                    "0.28 0.30");
            }

            rise_constraint ( mpw) {
                index_1("0.2, 0.4, 0.6");
                index_2("0.2, 0.4");
                values( "0.20 0.22", \
                    "0.24 0.26" \
                    "0.28 0.30");
            }
        }
    }

    ...
} /* end of arc */
} /* end of cell */
} /* end of library */

```

Checking min_pulse_width and minimum_period Constraints

The Library Compiler tool performs `min_pulse_width` and `minimum_period` constraint checks and issues an error message if the following conditions are not met:

- A two-dimensional `min_pulse_width` or `minimum_period` lookup table template must have the `related_out_total_output_net_capacitance` value defined in the index.

- The `related_output_pin` attribute must be specified on the clock pin that has the two-dimensional `min_pulse_width` or `minimum_period` lookup table with the additional output load index.

Using Conditional Attributes With No-Change Constraints

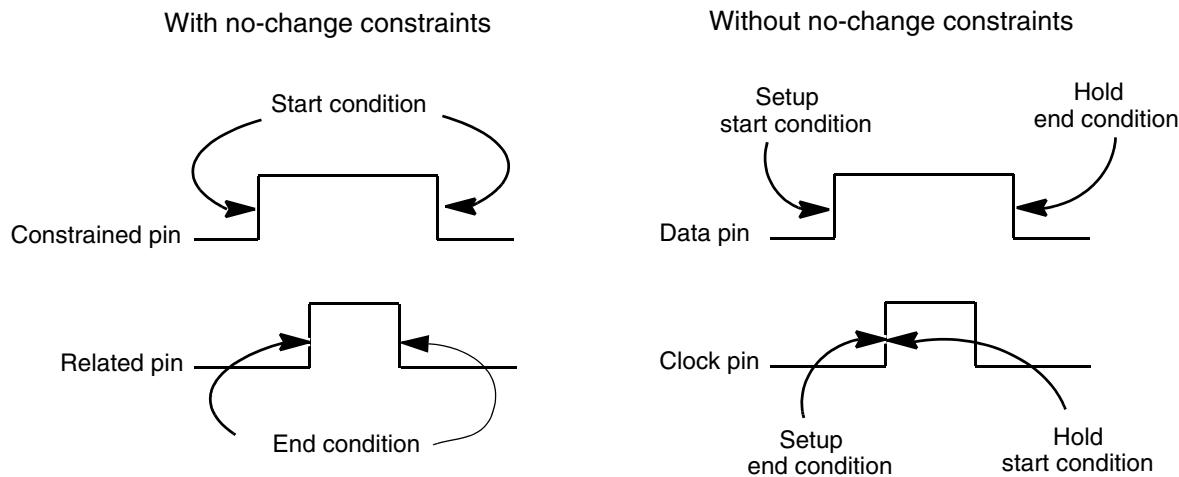
As shown in [Table 11-6](#), conditional timing check attributes have different interpretations when you use them with no-change timing constraints. See “[Setting No-Change Timing Constraints](#)” on page [11-66](#) for a description of no-change timing constraint values.

Table 11-6 Conditional Timing Attributes With No-Change Constraints

Conditional attributes	With no-change constraints
<code>when sdf_cond</code>	Defines both the constrained and related signal-enabling conditions
<code>when_start sdf_cond_start</code>	Defines the constrained signal-enabling condition
<code>when_end sdf_cond_end</code>	Defines the related signal-enabling condition
<code>sdf_edges : start_edge</code>	Adds edge specification to the constrained signal
<code>sdf_edges : end_edge</code>	Adds edge specification to the related signal
<code>sdf_edges : both_edges</code>	Specifies edges to both signals
<code>sdf_edges : noedges</code>	Specifies edges to neither signal

[Figure 11-19](#) shows the different interpretation of setup and hold conditions when used with a no-change timing constraint.

Figure 11-19 Interpretation of Conditional Timing Constraints With No-Change Constraints



Generating an SDF File

SDF file information applies to state-dependent delays as well as to conditional timing constraints.

You can generate an SDF file by first compiling your design in Design Compiler, and then using the `write_sdf` command to write out an SDF file. See the Synopsys man pages for more information about the `write_sdf` command.

An SDF file generated by Synopsys tools might contain conditional path delays. Create these path delays by putting the expression string defined in the `conditional_port_expr` attribute into the SDF file of the delay entry. When you generate a VHDL library, the string is written out for back-annotation reference, along with the delay value for the cell component.

When back-annotation is required for simulation, each `conditional_port_expr` in the SDF condition path delay is matched to a string in the VHDL library for an appropriate place to put the delay values that maintain the conditional path delay.

Note:

You must ensure that the string in the `sdf_cond` attribute, defined in the technology library, matches the `conditional_port_expr` statement of the corresponding state-dependent timing arc in SDF files. The Library Compiler tool ignores all white spaces during the string matching.

A common procedure is to use the Library Compiler tool to compile a file that contains one or more `sdf_cond` attributes. This creates a VHDL library containing state-dependent timing information that can be used for SDF condition matching by the Synopsys VHDL Simulation tool to match design rise and fall delays that third-party vendors might consider more

accurate. The intrinsic rise and fall delays defined in the library source file are ultimately replaced by the rise and fall delays defined in the design.

Impossible Transitions

This information applies to the table lookup delay model and only to combinational and three-state timing arcs. Certain output transitions cannot result from a single input change when `function`, `three_state`, and `x_function` share input.

In the following table, Y is the function of A, B, and C.

A	B	C	Y
0	0	0	Z
0	0	1	1
0	1	0	0
0	1	1	X
1	0	0	0
1	0	1	1
1	1	0	Z
1	1	1	X

No isolated signal change on C can cause the 0-to-1 or 1-to-0 transitions on Y. Therefore, there is no combinational arc from C to Y, although the two are functionally related. Further, no isolated change on A causes the 1-to-Z or Z-to-1 transitions on Y.

`three_state_enable` has no rising value (Z to 1), and `three_state_disable` has no falling value (1 to Z).

The Library Compiler tool detects such invalid timing arcs if they are specified. For the impossible 0-to-1 and 1-to-0 transitions, the tool deletes the timing arc. For the impossible 0-to-Z, Z-to-0, 1-to-Z, and Z-to-1 transitions, the tool marks the timing arc with the predefined `non_rising` or `non_falling` attribute.

Examples of NLDM Libraries

This section contains examples of libraries using the CMOS nonlinear delay model.

Library With Timing Constraints

In the nonlinear library description in [Example 11-12](#), pin D contains setup and hold constraints; pins Q and QN contain preset, clear, and edge-sensitive arcs.

Example 11-12 D Flip-Flop Description

```
library (NLDM) {
date : "January 14, 2015";
revision : 2015.01;
delay_model : table_lookup;
technology (cmos);
/* Define template of size 2 x 2*/
lu_table_template(cell_template) {
    variable_1 : input_net_transition;
    variable_2 : total_output_net_capacitance;
    index_1 ("0.0, 1.5");
    index_2 ("0.0, 4.0");
}
/* Define one-dimensional lu_table of size 4 */
lu_table_template(tran_template) {
    variable_1 : total_output_net_capacitance;
    index_1 ("0.0, 0.5, 1.5, 2.0");
}
cell( DFLOP_CLR_PRE ) {
    area : 11 ;
    ff ( IQ , IQN ) {
        clocked_on : " CLK ";
        next_state : " D ";
        clear : " CLR' ";
        preset : " PRE' ";
        clear_preset_var1 : L ;
        clear_preset_var2 : L ;
    }
    pin ( D ){
        direction : input;
        capacitance : 1 ;
        timing () {
            related_pin : "CLK" ;
            timing_type : hold_rising;
            rise_constraint(scalar) {
                values("0.12");
            }
            fall_constraint(scalar) {
                values("0.29");
            }
        }
    }
}
```

```
        }
    timing () {
        related_pin : "CLK" ;
        timing_type : setup_rising ;
        rise_constraint(scalar) {
            values("2.93");
        }
        fall_constraint(scalar) {
            values("2.14");
        }
    }
pin ( CLK ){
    direction : input;
    capacitance : 1 ;
}
pin ( PRE ) {
    direction : input ;
    capacitance : 2 ;
}
pin ( CLR ){
    direction : input ;
    capacitance : 2 ;
}
pin ( Q ) {
    direction : output;
    function : "IQ" ;

    timing () {
        related_pin : "PRE" ;
        timing_type : preset ;
        timing_sense : negative_unate ;
        cell_rise(cell_template) {
            values("0.00, 0.23", "0.11, 0.28");
        }
        rise_transition(tran_template) {
            values("0.01, 0.12, 0.15, 0.40");
        }
    }
    timing () {
        related_pin : "CLR" ;
        timing_type : clear ;
        timing_sense : positive_unate ;
        cell_fall(cell_template) {
            values("0.00, 0.24", "0.15, 0.26");
        }
        fall_transition(tran_template) {
            values("0.03, 0.15, 0.18, 0.38");
        }
    }
    timing () {
        related_pin : "CLK" ;
        timing_type : rising_edge;
```

```
cell_rise(cell_template) {
    values("0.00, 0.25", "0.11, 0.28");
}
rise_transition(tran_template) {
    values("0.01, 0.08, 0.15, 0.40");
}
cell_fall(cell_template) {
    values("0.00, 0.33", "0.11, 0.38");
}
fall_transition(tran_template) {
    values("0.01, 0.11, 0.18, 0.40");
}
}
pin ( QN ){
    direction : output ;
    function : "IQN" ;
    timing (){
        related_pin : "PRE" ;
        timing_type : clear ;
        timing_sense : positive_unate ;
        cell_fall(cell_template) {
            values("0.00, 0.23", "0.11, 0.28");
        }
        fall_transition(tran_template) {
            values("0.01, 0.12, 0.15, 0.40");
        }
    }
    timing (){
        related_pin : "CLR" ;
        timing_type : preset ;
        timing_sense : negative_unate ;
        cell_rise(cell_template) {
            values("0.00, 0.23", "0.11, 0.28");
        }
        rise_transition(tran_template) {
            values("0.01, 0.12, 0.15, 0.40");
        }
    }
    timing (){
        related_pin : "CLK" ;
        timing_type : rising_edge;
        cell_rise(cell_template) {
            values("0.00, 0.25", "0.11, 0.28");
        }
        rise_transition(tran_template) {
            values("0.01, 0.08, 0.15, 0.40");
        }
        cell_fall(cell_template) {
            values("0.00, 0.33", "0.11, 0.38");
        }
        fall_transition(tran_template) {
            values("0.01, 0.11, 0.18, 0.40");
        }
    }
}
```

```

        }
    }
}
}
```

CMOS Scalable Polynomial Delay Model

In the scalable polynomial delay model in [Example 11-13](#), pin D contains setup and hold constraints; pins Q and QN contain preset, clear, and edge-sensitive arcs.

Example 11-13 D Flip-Flop Description (CMOS Scalable Polynomial Delay Model)

```

library (SPDM) {
    technology (cmos);
    date : "September 19, 2002" ;
    revision : 2002.01 ;
    delay_model : polynomial ;
    /* Define template of 2D polynomial */
    poly_template(cell_template) {
        variables(input_net_transition, total_output_net_capacitance) ;
        variable_1_range(0.0, 1.5) ;
        variable_2_range(0.0, 4.0) ;
    }
    /* Define template of 1D polynomial */
    poly_template(tran_template) {
        variables(total_output_net_capacitance) ;
        variable_1_range(0.0, 2.0) ;
    }
    cell(DFLOP_CLR_PRE) {
        area : 11;
        ff(IQ, IQN) {
            clocked_on : "CLK" ;
            next_state : "D" ;
            clear : "CLR'" ;
            preset : "PRE'" ;
            clear_preset_var1 : L ;
            clear_preset_var2 : L ;
        }
        pin(D) {
            direction : input ;
            capacitance : 1 ;
            timing () {
                related_pin : "CLK" ;
                timing_type : hold_rising ;
                rise_constraint(scalar) {
                    values("0.12") ;
                }
                fall_constraint(scalar) {
                    values("0.29") ;
                }
            } /* end timing */
            timing () {
                related_pin : "CLK" ;
                timing_type : setup_rising ;
            }
        }
    }
}
```

```

rise_constraint(scalar) {
    values("2.93") ;
}
fall_constraint(scalar) {
    values("2.14") ;
}
} /* end timing */
} /* end pin D */
pin(CLK) {
    direction : input;
    capacitance : 1 ;
}
pin(PRE) {
    direction : input;
    capacitance : 2 ;
}
pin(CLR) {
    direction : input;
    capacitance : 2 ;
}
pin(Q) {
    direction : output ;
    function : "IQ" ;
    timing () {
        related_pin : "PRE" ;
        timing_type : preset ;
        timing_sense : negative_unate ;
        cell_rise(cell_template) {
            orders("1, 1") ;
            coefs("0.1632, 3.0688, 0.0013, 0.0320") ;
            variable_1_range (0.01, 3.00);
            variable_2_range (0.01, 3.00);
        }
        rise_transition(tran_template) {
            orders("1");
            coefs("0.2191, 1.7580") ;
            variable_1_range (0.01, 3.00);
            variable_2_range (0.01, 3.00);
        }
    } /* end timing */
    timing () {
        related_pin : "CLR" ;
        timing_type : clear ;
        timing_sense : positive_unate ;
        cell_fall(cell_template) {
            orders("1, 1") ;
            coefs("0.0542, 6.3294, 0.0214, -0.0310") ;
            variable_1_range (0.01, 3.00);
            variable_2_range (0.01, 3.00);
        }
        fall_transition(tran_template) {
            orders("1") ;
            coefs("0.0652, 2.9232") ;
            variable_1_range (0.01, 3.00);
            variable_2_range (0.01, 3.00);
        }
    } /* end timing */
    timing () {

```

```

related_pin : "CLK" ;
timing_type : rising_edge ;
cell_rise(cell_template) {
    orders("1, 1") ;
    coefs("0.1687, 3.0627, 0.0194, 0.0155") ;
    variable_1_range (0.01, 3.00) ;
    variable_2_range (0.01, 3.00) ;
}
rise_transition(tran_template) {
    orders("1");
    coefs("0.2130, 1.7576") ;
}
cell_fall(cell_template) {
    orders("1, 1") ;
    coefs("0.0539, 6.3360, 0.0194, -0.0289") ;
    variable_1_range (0.01, 3.00) ;
    variable_2_range (0.01, 3.00) ;
}
fall_transition(tran_template) {
    orders("1");
    coefs("0.0647, 2.9220") ;
    variable_1_range (0.01, 3.00) ;
    variable_2_range (0.01, 3.00) ;
}
} /* end timing */
} /* end pin Q */
pin(QN) {
    direction : output ;
    function : "IQN" ;
    timing () {
        related_pin : "PRE" ;
        timing_type : clear ;
        timing_sense : positive_unate ;
        cell_fall(cell_template) {
            orders ("1, 1");
            coefs("0.1605, 3.0639, 0.0325, 0.0104") ;
            variable_1_range (0.01, 3.00) ;
            variable_2_range (0.01, 3.00) ;
        }
        fall_transition(tran_template) {
            orders ("1") ;
            coefs("0.1955, 1.7535") ;
            variable_1_range (0.01, 3.00) ;
            variable_2_range (0.01, 3.00) ;
        }
    } /* end timing */
    timing () {
        related_pin : "CLR" ;
        timing_type : preset ;
        timing_sense : negative_unate ;
        cell_rise(cell_template) {
            orders ("1, 1");
            coefs("0.0540, 6.3849, 0.0211, -0.0720" );
            variable_1_range (0.01, 3.00) ;
            variable_2_range (0.01, 3.00) ;
        }
        rise_transition(tran_template) {
            orders ("1") ;
        }
    }
}

```

```

        coefs("0.0612, 2.9541") ;
variable_1_range (0.01, 3.00);
variable_2_range (0.01, 3.00);
}
} /* end timing */
timing () {
    related_pin : "CLK" ;
    timing_type : rising_edge ;
    cell_rise(cell_template) {
        orders ("1, 1") ;
        coefs ("0.2407, 3.1568, 0.0129, 0.0143") ;
        variable_1_range (0.01, 3.00);
        variable_2_range (0.01, 3.00);
    }
    rise_transition(tran_template) {
        orders ("1") ;
        coefs ("0.3355, 1.7578") ;
        variable_1_range (0.01, 3.00);
        variable_2_range (0.01, 3.00);
    }
    cell_fall(cell_template) {
        orders("1, 1") ;
        coefs("0.0742, 6.3452, 0.0260, -0.0938") ;
        variable_1_range (0.01, 3.00);
        variable_2_range (0.01, 3.00);
    }
    fall_transition(tran_template) {
        orders ("1") ;
        coefs("0.0597, 2.9997") ;
        variable_1_range (0.01, 3.00);
        variable_2_range (0.01, 3.00);
    }
}
} /* end timing */
} /* end pin QN */
} /* end library */

```

Library With Clock Insertion Delay

```

library( vendor_a ) {
delay_model :table_lookup;
/* 1. Define library-level one-dimensional lu_table of size 4 */
lu_table_template(lu_template) {
    variable_1 :input_net_transition;
    index_1 ("0.0, 0.5, 1.5, 2.0");
}
/* 2. Define a cell and pins within it which has clock tree path*/
cell (general) {
...
pin(clk) {
    direction:input;
    timing() {
        timing_type :max_clock_tree_path;
        timing_sense :positive_unate;
        cell_rise(lu_template) {
            values ("0.1, 0.15, 0.20, 0.29");
        }
    }
}

```

```

        cell_fall(lu_template) {
            values ("0.2, 0.25, 0.30, 0.39");
        }
    }
    timing() {
        timing_type :min_clock_tree_path;
        timing_sense :positive_unate;
        cell_rise(lu_template) {
            values ("0.2, 0.35, 0.40, 0.59");
        }
        cell_fall(lu_template) {
            values ("0.3, 0.45, 0.50, 0.69");
        }
    }
}
...
}

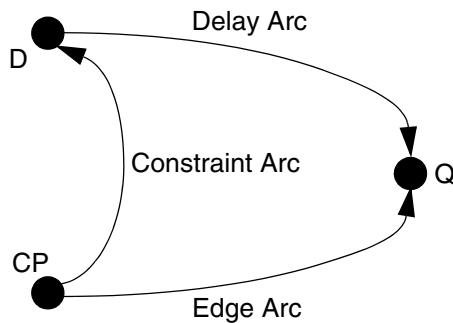
```

Describing a Transparent Latch Clock Model

The `tLatch` group lets you specify a functional latch when the latch arcs are absent. You use the `tLatch` group at the data pin level to specify the relationship between the data pin and the enable pin on a latch.

[Figure 11-20](#) shows a transparent latch timing model.

Figure 11-20 Transparent Latch Timing Model



Syntax

```

library (name_string) {
    cell (name_string) {
        ...
        timing_model_type : value_enum ;
        ...
        pin (data_pin_name_string) {
            tLatch (enable_pin_name_string) {
                edge_type : value_enum ;

```

```
        tdisable : valueBoolean ;  
    }  
}  
}  
}
```

The `tLatch` name specifies the enable pin that defines the latch clock pin. You define the `tLatch` group in a `pin` group, but it is only effective if you also define the `timing_model_type` attribute in the cell that the pin belongs to. The `timing_model_type` attribute can have the following values: abstracted, extracted, and quick timing model.

A `tLatch` group is optional. You can define one or more `tLatch` groups for a pin, but you must not define more than two `tLatch` groups between the same pair of data and enable pins, one rising and one falling. Also, the data pin and the enable pin must be different.

Pins in the `tLatch` group can be input or inout pins or internal pins. When a `tLatch` group is not present, a timing analysis tool infers the latch clock pin based on the presence of:

- A `related_pin` statement in a timing group with either a `rising_edge` or `falling_edge` `timing_type` value within a latch output pin
 - A `related_pin` statement in a timing group with a `setup`, `hold_rising`, or `falling` `timing_type` value within a latch input pin

The `edge_type` attribute defines whether the latch is positive (high) transparent or negative (low) transparent. The rising and falling `edge_type` values specify the opening edge, and therefore the transparent window of the latch, and completely define the latch to be level-high transparent or level-low transparent.

When a `tLatch` group is not present, a timing analysis tool infers transparency on an output pin based on the timing arc attribute and the presence of a latch functional construct on that pin.

The rising and falling edge type attribute values explicitly define the transparency windows

- When the rising_edge and falling_edge timing_type values are missing
 - When the rising_edge and falling_edge timing_type values are different from the latch transparency

The `tdisable` attribute disables transparency in a latch. During path propagation, timing analysis tools disable and ignore all data pin output pin arcs that reference a `tLatch` group whose `tdisable` attribute is set to true on an edge triggered flip-flop.

Example

```
pin (D) {  
    tlatch (CP) {  
        edge_type : rising ;  
        tdisable : true ;
```

```
        }
    }
pin (Q) {
    direction : output ;
    timing () {
        timing_sense : positive_unate ;
        related_pin : D ;
        cell_rise ...
    }
    timing () {
        /* optional arc that can differ from edge_type */
        timing_type : falling_edge ;
        related_pin : CP ;
        cell_fall ...
    }
}
```

Checking Timing and Nonlinear Delay Data

You can use the NLDM library screener to check for timing data and nonlinear delay errors in your library models. For information about using the NLDM library screener and for descriptions of the nonlinear delay noise variables, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Driver Waveform Support

In cell characterization, the shape of the waveform driving the characterized circuit can have a significant impact on the final results. Typically, the waveform is generated by a simple piecewise linear (PWL) waveform or an active-driver cell (a buffer or inverter).

Liberty supports driver waveform syntax, which specifies the type of waveform that is applied to library cells during characterization. The driver waveform syntax helps facilitate the characterization process for existing libraries and correlation checking. The driver waveform requirements can vary.

Possible usage models include:

- Using a common driver waveform for all cells.
- Using a different driver waveform for different categories of cells.
- Using a pin-specific driver waveform for complex cell pins.
- Using a different driver waveform for rise and fall timing arcs.

Syntax

The driver waveform syntax is as follows:

```
library(library_name) {
    ...
    lu_table_template (waveform_template_name) {
        variable_1: input_net_transition;
        variable_2: normalized_voltage;
        index_1 ("float..., float");
        index_2 ("float..., float");
    }
    normalized_driver_waveform(waveform_template_name) {
        driver_waveform_name : string; /* Specifies the name of the driver
                                         waveform table */
        index_1 ("float..., float"); /* Specifies input net transition */
        index_2 ("float..., float"); /* Specifies normalized voltage */
        values ( "float..., float", \ /* Specifies the time in library units */
                 ...
                 "float..., float");
    }
    ...
    cell (cell_name) {
        ...
        driver_waveform : string;
        driver_waveform_rise : string;
        driver_waveform_fall : string;
        pin (pin_name) {
            driver_waveform : string;
            driver_waveform_rise : string;
            driver_waveform_fall : string;
            ...
        }
    }
}/* end of library*/
```

In the driver waveform syntax, the first index value in the table specifies the input slew and the second index value specifies the voltage normalized to VDD. The values in the table specify the time in library units (not scaled) when the waveform crosses the corresponding voltages. The `driver_waveform_name` attribute specified for the driver waveform table differentiates the tables when multiple driver waveform tables are defined.

The cell-level `driver_waveform`, `driver_waveform_rise` and `driver_waveform_fall` attributes meet cell-specific and rise- and fall-specific requirements. The attributes refer to the driver waveform table name predefined at the library level.

Similar to the cell-level driver waveform attributes, the pin group includes the `driver_waveform`, `driver_waveform_rise` and `driver_waveform_fall` attributes to meet pin-specific predriver requirements for complex cell pins (such as macro cells). These attributes also refer to the predefined driver waveform table name.

Library-Level Tables, Attributes, and Variables

This section describes driver waveform tables, attributes, and variables that are specified at the library level.

normalized_voltage Variable

The `normalized_voltage` variable is specified under the `lu_table_template` table to describe a collection of waveforms with various input slew values. For a given input slew in `index_1` (for example, `index_1[0] = 1.0 ns`), the `index_2` values are a set of points that represent how the voltage rises from 0 to VDD in a rise arc, or from VDD to 0 in a fall arc.

Rise Arc Example

```
normalized_driver_waveform (waveform_template) {
    index_1 ("1.0"); /* Specifies the input net transition*/
    index_2 ("0, 0.1, 0.3, 0.5, 0.7, 0.9, 1.0"); /* Specifies
        the voltage normalized to VDD */
    values ("0, 0.2, 0.4, 0.6, 0.8, 0.9, 1.1"); /* Specifies the
        time when the voltage reaches the index_2 values*/
}
```

The `lu_table_template` table represents an input slew of 1.0 ns, when the voltage is 0%, 10%, 30%, 50%, 70%, 90% or 100% of VDD, and the time values are 0, 0.2, 0.4, 0.6, 0.8, 0.9, 1.1 (ns). The time value can go beyond the corresponding input slew because a long tail might exist in the waveform before it reaches the final status.

normalized_driver_waveform Group

The library-level `normalized_driver_waveform` group represents a collection of driver waveforms under various input slew values. The `index_1` specifies the input slew and `index_2` specifies the normalized voltage.

The slew index in the `normalized_driver_waveform` table is based on the slew derate and slew trip points of the library (global values). When applied on a pin or cell with different slew or slew derate, the new slew should be interpreted from the waveform.

driver_waveform_name Attribute

The `driver_waveform_name` string attribute differentiates the driver waveform table from other driver waveform tables when multiple tables are defined. Cell-specific, rise-specific, and fall-specific driver waveform usage modeling depend on this attribute.

The `driver_waveform_name` attribute is optional. You can define a driver waveform table without the attribute, but there can be only one table in a library, and that table is regarded as the default driver waveform table for all cells in the library. If more than one table is defined without the attribute, the last table is used. The other tables are ignored and not stored in the library database.

Cell-Level Attributes

This section describes driver waveform attributes defined at the cell level.

driver_waveform Attribute

The `driver_waveform` attribute is an optional string attribute that allows you to define a cell-specific driver waveform. The value must be the `driver_waveform_name` predefined in the `normalized_driver_waveform` table. Otherwise, the Library Compiler tool issues an error.

When the attribute is defined, the cell uses the specified driver waveform during characterization. When it is not specified, the common driver waveform (the `normalized_driver_waveform` table without the `driver_waveform_name` attribute) is used for the cell.

driver_waveform_rise and driver_waveform_fall Attributes

The `driver_waveform_rise` and `driver_waveform_fall` string attributes are similar to the `driver_waveform` attribute. These two attributes allow you to define rise-specific and fall-specific driver waveforms. The `driver_waveform` attribute can coexist with the `driver_waveform_rise` and `driver_waveform_fall` attributes, though the `driver_waveform` attribute becomes redundant.

You should specify a driver waveform for a cell by using the following priority:

1. Use the `driver_waveform_rise` for a rise arc and the `driver_waveform_fall` for a fall arc during characterization. If they are not defined, specify the second and third priority driver waveforms.
2. Use the cell-specific driver waveform (defined by the `driver_waveform` attribute).
3. Use the library-level default driver waveform (defined by the `normalized_driver_waveform` table without the `driver_waveform_name` attribute).

The `driver_waveform_rise` attribute can refer to a `normalized_driver_waveform` that is either rising or falling. You can invert the waveform automatically during runtime if necessary.

Pin-Level Attributes

This section describes driver waveform attributes defined at the pin level.

driver_waveform Attribute

The `driver_waveform` attribute is the same as the `driver_waveform` attribute specified at the cell level. For more information, see “[driver_waveform Attribute](#)” on page 11-93.

driver_waveform_rise and driver_waveform_fall Attributes

The `driver_waveform_rise` and `driver_waveform_fall` attributes are the same as the `driver_waveform_rise` and `driver_waveform_fall` attributes specified at the cell level. For more information, see “[driver_waveform_rise and driver_waveform_fall Attributes](#)” on page 11-93.

Checking Driver Waveform Data

The Library Compiler tool checks the `normalized_driver_waveform` table data at the library level and cell level. For information about the types of checks the Library Compiler tool performs, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Driver Waveform Example

```
library(test_library) {  
  
    lu_table_template(waveform_template) {  
        variable_1 : input_net_transition;  
        variable_2 : normalized_voltage;  
        index_1 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7");  
        index_2 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");  
    }  
  
    /* Specifies the default library-level driver waveform table (the  
       default driver waveform without the driver_waveform attribute) */  
    normalized_driver_waveform (waveform_template) {  
        values ("0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09", \  
                "0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19", \  
                ...  
                "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");  
    }  
    /* Specifies the driver waveform for the clock pin */  
    normalized_driver_waveform (waveform_template) {  
        driver_waveform_name : clock_driver;  
    }  
}
```

```

index_1 ("0.15, 0.25, 0.35, 0.45, 0.55, 0.65, 0.75");
index_2 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
values ("0.012, 0.03, 0.045, 0.06, 0.075, 0.090, 0.105, 0.13,
0.145", \
        ...
        "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
}
/* Specifies the driver waveform for the bus */
normalized_driver_waveform (waveform_template) {
    driver_waveform_name : bus_driver;
    index_1 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7");
    index_2 ("0.15, 0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85, 0.95");
    values ("0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09", \
            "0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19", \
            ...
            "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
}
/* Specifies the driver waveform for the rise */
normalized_driver_waveform (waveform_template) {
    driver_waveform_name : rise_driver;
    index_1 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7");
    index_2 ("0.15, 0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85, 0.95");
    values ("0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09", \
            "0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19", \
            ...
            "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
}
/* Specifies the driver waveform for the fall */
normalized_driver_waveform (waveform_template) {
    driver_waveform_name : fall_driver;
    index_1 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7");
    index_2 ("0.15, 0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85, 0.95");
    values ("0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09", \
            "0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19", \
            ...
            "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
}
...
cell (my_cell1) {
    driver_waveform : clock_driver;
    ...
}
cell (my_cell2) {
    driver_waveform : bus_driver;
    ...
}
cell (my_cell3) {
    driver_waveform_rise : rise_driver;
    driver_waveform_fall : fall_driver;
    ...
}
cell (my_cell4) {
/* No driver_waveform attribute is specified. Use the default driver

```

```
    waveform */
...
}
/* End library */
```

Sensitization Support

Timing information specified in libraries results from circuit simulator (library characterization) tools. The models themselves often represent only a partial record of how a particular arc was sensitized during characterization. To fully reproduce the conditions that allowed the model to be generated, the states of all the input pins on the cell must be known.

In composite current source (CCS) models, the accuracy requirements are very high (the expectation is as high as 2 percent compared to SPICE). To achieve this level of accuracy, correlation with SPICE requires that the cell conditions be represented exactly as during characterization. For more information about CCS models, see [Chapter 14, “Composite Current Source Modeling.”](#)

The following sections describe the pin sensitization condition information details used to characterize the timing data. The syntax predeclares state vectors as reusable sensitization patterns in the library. The patterns are referenced and instantiated as stimuli waveforms specific to timing arcs. The same sensitization pattern can be referenced by multiple cells or multiple timing arcs. One cell can also reference multiple sensitization patterns, which saves storage resources. The Liberty attributes and groups highlighted in the following sections help to specify the sensitization information of timing arcs during simulation and characterization.

sensitization Group

The `sensitization` group defined at the library level describes the complete state patterns for a specific list of pins (defined by the `pin_names` attribute) that is referenced and instantiated as stimuli in the timing arc.

Vector attributes in the group define all possible pin states used as stimuli. Actual stimulus waveforms can be described by a combination of these vectors. Multiple `sensitization` groups are allowed in a library. Each `sensitization` group can be referenced by multiple cells, and each cell can make reference to multiple `sensitization` groups.

The following attributes are library-level attributes under the `sensitization` group.

pin_names Attribute

The `pin_names` complex attribute defines a default list of pin names. All vectors in this sensitization group are the exhaustive list of all possible transitions of the input pins and their subsequent output response.

The `pin_names` attribute is required, and it must be declared in the sensitization group before all `vector` declarations.

vector Attribute

Similar to the `pin_names` attribute, the `vector` attribute describes a transition pattern for the specified pins. The stimulus is described by an ordered list of vectors.

The two arguments for the `vector` attribute are as follows:

`vector id`

The `vector id` argument is an identifier to the vector string. The `vector id` value must be an integer greater than or equal to zero and unique among all vectors in the current sensitization group.

`vector string`

The `vector string` argument represents a pin transition state. The string consists of the following transition status values: 0, 1, X, and Z where each character is separated by a space. The number of elements in the vector string must equal the number of arguments in `pin_names`.

The `vector` attribute can also be declared as:

```
vector (positive_integer, "[0|1|X|Z] [0|1|X|Z]...");
```

Example

```
sensitization(sensitization_nand2) {
    pin_names ( IN1, IN2, OUT1 );
    vector ( 1, "0 0 1" );
    vector ( 2, "0 1 1" );
    vector ( 3, "1 0 1" );
    vector ( 4, "1 1 0" );
```

Cell-Level Attributes

Generally, one cell references one sensitization group for cells. A cell-level attribute that can link the cell with a specific sensitization group helps to simplify sensitization usage. The following cell-level attributes ensure that sensitization groups can be referenced by cells with similar functionality but can have different pin names.

sensitization_master Attribute

The `sensitization_master` attribute defines the sensitization group referenced by the cell to generate stimuli for characterization. The attribute is required if the cell contains sensitization information. Its string value should be any `sensitization` group name predefined in the current library.

pin_name_map Attribute

The `pin_name_map` attribute defines the pin names that are used to generate stimuli from the `sensitization` group for all timing arcs in the cell. The `pin_name_map` attribute is optional when the pin names in the cell are the same as the pin names in the sensitization master, but it is required when they are different.

If the `pin_name_map` attribute is set, the number of pins must be the same as that in the sensitization master, and all pin names should be legal pin names in the cell.

timing Group Attributes

This section describes the `sensitization_master` and `pin_name_map` timing-arc attributes. These attributes enable a complex cell (a macro in most cases) to refer to multiple sensitization groups. You can also specify a sampling vector and user-defined time intervals between vectors. The `wave_rise` and `wave_fall` attributes, which describe characterization stimuli (instantiated in pin timing arcs), are also discussed in this section.

sensitization_master Attribute

The `sensitization_master` simple attribute defines the sensitization group specific to the current timing group to generate stimulus for characterization. The attribute is optional when the sensitization master used for the timing arc is the same as that defined in the current cell, and it is required when they are different. Any sensitization group name predefined in the current library is a valid attribute value.

pin_name_map Attribute

Similar to the `pin_name_map` attribute defined in the cell level, the timing-arc `pin_name_map` attribute defines pin names used to generate stimulus for the current timing arc. The attribute is optional when `pin_name_map` pin names are the same as the following (listed in order of priority):

1. Pin names in the `sensitization_master` of the current timing arc.
2. Pin names in the `pin_name_map` attribute of the current cell group.
3. Pin names in the `sensitization_master` of the current cell group.

The `pin_name_map` attribute is required when `pin_name_map` pin names are different from all of the pin names in the previous list.

wave_rise and wave_fall Attributes

The `wave_rise` and `wave_fall` attributes represent the two stimuli used in characterization. The value for both attributes is a list of integer values, and each value is a vector ID predefined in the library sensitization group. The following example describes the `wave_rise` and `wave_fall` attributes:

```
wave_rise (vector_id[m]..., vector_id[n]);
wave_fall (vector_id[j]..., vector_id[k]);
```

Example

```
library(my_library) {
    ...
    sensitization(sensi_2in_lout) {
        pin_names (IN1, IN2, OUT);
        vector (0, "0 0 0");
        vector (1, "0 0 1");
        vector (2, "0 1 0");
        vector (3, "0 1 1");
        vector (4, "1 0 0");
        vector (5, "1 0 1");
        vector (6, "1 1 0");
        vector (7, "1 1 1");
    }
    cell (my_nand2) {
        sensitization_master : sensi_2in_lout;
        pin_name_map (A, B, Z); /* these are pin names for the sensitization
                               in this cell. */
        ...
        pin(A) {
            ...
        }
        Pin(B) {
            ...
        }
        pin(Z) {
            ...
            timing() {
                related_pin : "A";
                wave_rise (6, 3);
            /* 6, 3 - vector id in sensi_2in_lout sensitization group. Waveform
               interpretation of wave_rise is (for "A, B, Z" pins): 10 1 01 */
                wave_fall (3, 6);
            ...
        }
        timing() {
            related_pin : "B";
            wave_rise (7, 4); /* 7, 4 - vector id in sensi_2in_lout
                               sensitization group. Waveform interpretation of
                               wave_rise is (for "A, B, Z" pins): 01 1 10 */
        }
    }
}
```

```

        sensitization group. */

    wave_fall (4, 7);
    ...
}
} /* end pin(Z) */
} /* end cell(my_nand2) */
...
} /* end library */

```

wave_rise_sampling_index and wave_fall_sampling_index Attributes

The `wave_rise_sampling_index` and `wave_fall_sampling_index` simple attributes override the default behavior of the `wave_rise` and `wave_fall` attributes. (The `wave_rise` and `wave_fall` attributes select the first and the last vectors to define the sensitization patterns of the input to the output pin transition that are predefined inside the sensitization template specified at the library level).

By default, the Library Compiler tool assumes that the delay is measured from the last transition of the sensitization description to the transition of the output pin.

Example

```
wave_rise (2, 5, 7, 6); /* wave_rise ( wave_rise[0],
wave_rise[1], wave_rise[2], wave_rise[3] ); */
```

In the previous example, the wave rise vector delay is measured from the last transition (vector 7 changing to vector 6) to the output transition. The default `wave_rise_sampling_index` value is the last entry in the vector, which is 3 in this case (because the numbering begins at 0).

To override this default, set the `wave_rise_sampling_index` attribute, as shown:

```
wave_rise_sampling_index : 2 ;
```

When you specify this attribute, the delay is measured from the second last transition of the sensitization vector to the final output transition, in other words from the transition of vector 5 to vector 7.

Note:

You cannot specify a value of 0 for the `wave_rise_sampling_index` attribute.

wave_rise_time_interval and wave_fall_time_interval Attributes

The `wave_rise_time_interval` and `wave_fall_time_interval` attributes control the time interval between transitions. By default, the stimuli (specified in `wave_rise` and `wave_fall`) are widely spaced apart during characterization (for example, 10 ns from one

vector to the next) to allow all output transitions to stabilize. The attributes allow you to specify a short duration between one vector to the next to characterize special purpose cells and pessimistic timing characterization.

The `wave_rise_time_interval` and `wave_fall_time_interval` attributes are optional when the default time interval is used for all transitions, and they are required when you need to define special time intervals between transitions. Usually, the special time interval is smaller than the default time interval.

The `wave_rise_time_interval` and `wave_fall_time_interval` attributes can have an argument count from 1 to $n-1$, where n is the number of arguments in corresponding `wave_rise` or `wave_fall`. Use 0 to imply the default time interval used between vectors.

Example

```
wave_rise (2, 5, 7, 6); /* wave_rise ( wave_rise[0],  
wave_rise[1], wave_rise[2], wave_rise[3] ); */  
wave_rise_time_interval (0.0, 0.3);
```

The previous example suggests the following:

- Use the default time interval between `wave_rise[0]` and `wave_rise[1]` (in other words, vector 2 and vector 5).
- Use 0.3 between `wave_rise[1]` and `wave_rise[2]` (in other words, vector 5 and vector 7).
- Use the default time interval between `wave_rise[2]` and `wave_rise[3]` in other words, vector 7 and vector 6).

timing Group Syntax

```
library(library_name) {  
    ...  
    sensitization (sensitization_group_name) {  
        pin_names (string..., string);  
        vector (integer, string);  
        ...  
        vector (integer, string);  
    }  
    ...  
  
    cell(cell_name) {  
        sensitization_master : sensitization_group_name;  
        pin_name_map (string..., string);  
        ...  
        pin(pin_name) {  
            ...  
            timing() {
```

```
related_pin : string;
sensitization_master : sensitization_group_name;
pin_name_map (string,..., string);
wave_rise (integer,..., integer);
wave_fall (integer,..., integer);
wave_rise_sampling_index : integer;
wave_fall_sampling_index : integer;
wave_rise_timing_interval (float..., float);
wave_fall_timing_interval (float..., float);
...
}/*end of timing */
}/*end of pin */
}/*end of cell */
...
}/* end of library*/
```

Checking Sensitization Data

The Library Compiler tool checks sensitization information and reports warnings and errors if it encounters issues. For information about the types of sensitization checks the Library Compiler tool performs, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

NAND Cell Example

The following is an example of a NAND cell with sensitization information.

```
library(cell1) {
    sensitization(sensitization_nand2) {
        pin_names ( IN1, IN2, OUT1 );
        vector ( 1, "0 0 1" );
        vector ( 2, "0 1 1" );
        vector ( 3, "1 0 1" );
        vector ( 4, "1 1 0" );
    }

    cell (nand2) {
        sensitization_master : sensitization_nand2;
        pin_name_map (A, B, Y);
        pin (A) {
            direction : input ;
            ...
        }
        pin (B) {
            direction : input ;
            ...
        }
        pin (Y) {
            direction : output;
```

```

...
timing() {
    related_pin : "A";
    timing_sense : negative_unate;
    wave_rise ( 4, 2 ); /* 10 1 01 */
    wave_fall ( 2, 4 ); /* 01 1 10 */
    ...
}
timing() {
    related_pin : "B";
    timing_sense : negative_unate;
    wave_rise ( 4, 3 );
    wave_fall ( 3, 4 );
    ...
}
} /* end pin(Y) */
} /* end cell(nand2) */
} /* end library */

```

Complex Macro Cell Example

The following is an example of a complex macro cell highlighting the usage of the `sensitization_master` group inside the `timing` group.

```

library(cell1) {
    sensitization(sensitization_2in_1out) {
        pin_names ( IN1, IN2, OUT );
        vector ( 1, "0 0 1" );
        vector ( 2, "0 1 1" );
        vector ( 3, "1 0 1" );
        vector ( 4, "1 1 0" );
    }
    sensitization(sensitization_3in_1out) {
        pin_names ( IN1, IN2, IN3, OUT );
        vector ( 0, "0 0 0 0" );
        vector ( 1, "0 0 0 1" );
        vector ( 2, "0 0 1 0" );
        vector ( 3, "0 0 1 1" );
        vector ( 4, "0 1 0 0" );
        vector ( 5, "0 1 0 1" );
        vector ( 6, "0 1 1 0" );
        vector ( 7, "0 1 1 1" );
        vector ( 8, "1 0 0 0" );
        vector ( 9, "1 0 0 1" );
        vector ( 10, "1 0 1 0" );
        vector ( 11, "1 0 1 1" );
        vector ( 12, "1 1 0 0" );
        vector ( 13, "1 1 0 1" );
        vector ( 14, "1 1 1 0" );
        vector ( 15, "1 1 1 1" );
    }
}

```

```
cell (nand2) {
    sensitization_master : sensitization_2in_1out;
    pin_name_map (A, B, Y);
    pin (A) {
        direction : input ;
        ...
    }
    pin (B) {
        direction : input ;
        ...
    }
    pin (CIN0) {
        direction : input ;
        ...
    }
    pin (CIN1) {
        direction : input ;
        ...
    }
    pin (CK) {
        direction : input;
        ...
    }
    pin (Y) {
        direction : output;
        ...
        timing() {
            related_pin : "A";
            /* inherit sensitization_master & pin_name_map from cell level */
            timing_sense : negative_unate;
            wave_rise ( 4, 2 );
            wave_fall ( 2, 4 );
            ...
        }
        timing() {
            related_pin : "B";
            timing_sense : negative_unate;
            wave_rise ( 4, 3 );
            wave_fall ( 3, 4 );
            ...
        }
    } /* end pin(Y) */
    pin (Z) {
        direction : output;
        ...
        timing () {
            related_pin : "CK";
            sensitization_master : sensitization_3in_1out; /* timing arc
specific sensitization master, overwrite the cell level attribute. */

            pin_name_map (CIN0, CIN1, CK, Z); /* timing arc specific
pin_name_map, overwrite the cell level attribute. */
        }
    }
}
```

```
        wave_rise (14, 4, 0, 3, 10, 5); /* the waveform describe here
has no real meaning, just select random vector id in sensitization
sensitization_3in_1out group. */

        wave_fall (15, 9, 3, 1, 6, 7);
        wave_rise_sampling_index : 3; /* sampling index, specific for
this timing arc. */

        wave_fall_sampling_index : 3;
        wave_rise_timing_interval(0, 0.3, 0.3); /* special timing
interval, specific for this timing arc. */

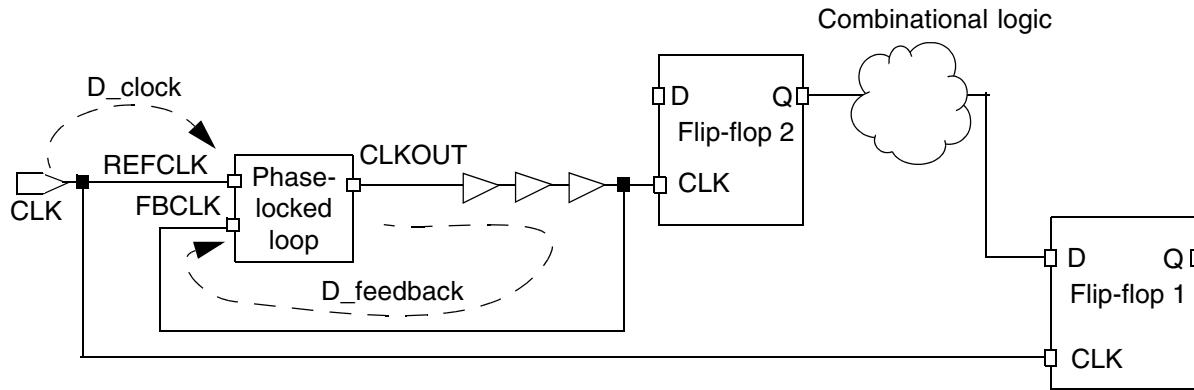
        wave_fall_timing_interval(0, 0.3, 0.3);
    }
} /* end pin (Z) */
} /* end cell(nand2) */
} /* end library */
```

Phase-Locked Loop Support

A phase-locked loop (PLL) is a feedback control system that automatically adjusts the phase of a locally-generated signal to match the phase of an input signal. Phase-locked loops contain the following pins:

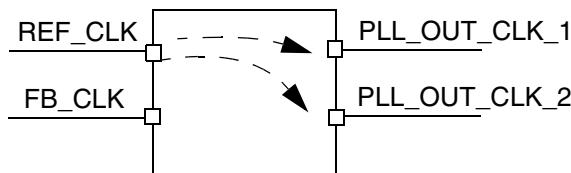
- The reference clock pin where the reference clock is connected
- The phase-locked loop output clock pin where the phase-locked loop generates a phase-shifted version of the reference clock
- The feedback pin where the feedback path from the output of the clock ends

[Figure 11-21](#) shows a circuit with a phase-locked loop. Without the phase-locked loop block, there is a large clock skew in the clocks arriving at the launch point and at the flip-flops. This large skew results in an extremely tight delay constraint for the combinational logic. A phase-locked loop reduces the large skew between the launch point and the flip-flops.

Figure 11-21 Phase-Locked Loop Circuit

The phase-locked loop generates a phase-shifted version of the reference clock at the output pin so that the phase of the feedback clock matches the phase of the reference clock. If the clock arrives at the reference pin of the phase-locked loop at the time, *D_clock*, and the delay of the feedback path is *D_feedback*, the source latency of the clock generated at the output of the phase-locked loop is *D_clock* - *D_feedback*. The clock arrives at the feedback pin at time, *D_clock*, which is the same as the time the clock arrives at the reference pin. Therefore, the phase-locked loop eliminates the latency on the launch path and relaxes the delay constraint on the combinational logic.

[Figure 11-22](#) shows a simple phase-locked loop model. The phase-locked loop information that a library should contain are pins and timing arcs inside the phase-locked loop. The forward arcs from REF_CLK to PLL_OUT_CLK_* in the figure simulate the phase shift behavior of the phase-locked loop. There are two half-unate arcs from the reference clock to each output of the phase-locked loop.

Figure 11-22 Simple Phase-Locked Loop Model

Phase-Locked Loop Syntax

```
cell (cell_name) {
    is_pll_cell : true;
    pin (ref_pin_name) {
        is_pll_reference_pin : true;
        direction : output;
        ...
    }
}
```

```
pin (feedback_pin_name) {
    is_pll_feedback_pin : true;
    direction : output;
    ...
}
pin (output_pin_name) {
    is_pll_output_pin : true;
    direction : output;
    ...
}
}
```

Cell-Level Attribute

This section describes a cell-level attribute.

is_pll_cell Attribute

The `is_pll_cell` Boolean attribute identifies a phase-locked loop cell.

Pin-Level Attributes

This section describes pin-level attributes.

is_pll_reference_pin Attribute

The `is_pll_reference_pin` Boolean attribute tags a pin as a reference pin on the phase-locked loop. In a phase-locked loop cell group, the `is_pll_reference_pin` attribute should be set to true in only one input pin group.

is_pll_feedback_pin Attribute

The `is_pll_feedback_pin` Boolean attribute tags a pin as a feedback pin on a phase-locked loop. In a phase-locked loop cell group, the `is_pll_feedback_pin` attribute should be set to true in only one input pin group.

is_pll_output_pin Attribute

The `is_pll_output_pin` Boolean attribute tags a pin as an output pin on a phase-locked loop. In a phase-locked loop cell group, the `is_pll_output_pin` attribute should be set to true in one or more output pin groups.

Phase-Locked Loop Example

```
cell(my_pll) {
    is_pll_cell : true;

    pin( REFCLK ) {
        direction : input;
        is_pll_reference_pin : true;
    }

    pin( FBKCLK ) {
        direction : input;
        is_pll_feedback_pin : true;
    }

    pin (OUTCLK_1x) {
        direction : output;
        is_pll_output_pin : true;
        timing() /* Timing Arc */
            related_pin: "REFCLK";
            timing_type: combinational_rise;
            timing_sense: positive_unate;
        . . .

    }
    timing() /* Timing Arc */
        related_pin: "REFCLK";
        timing_type: combinational_fall;
        timing_sense: positive_unate;
    . . .
}
}

pin (OUTCLK_2x) {
direction : output;
is_pll_output_pin : true;
timing() /* Timing Arc */
    related_pin: "REFCLK";
    timing_type: combinational_rise;
    timing_sense: positive_unate;
. . .
}
timing() /* Timing Arc */
    related_pin: "REFCLK";
    timing_type: combinational_fall;
    timing_sense: positive_unate;
. . .
}
} /* End pin group */
} /* End cell group */
```

Checking Phase-Locked Loop Libraries

The Library Compiler tool automatically checks phase-locked loop libraries and reports any problems it encounters. For information about the types of checks the Library Compiler tool performs for phase-locked loop libraries, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

12

Modeling Power and Electromigration

This chapter provides an overview of modeling static and dynamic power for CMOS technology.

To model CMOS static and dynamic power, you must understand the topics covered in the following sections:

- Modeling Power Terminology
- Switching Activity
- Modeling Cells for Power
- Modeling for Leakage Power
- Representing Leakage Power Information
- Threshold Voltage Modeling
- Modeling for Internal and Switching Power
- Representing Internal Power Information
- Defining Internal Power Groups
- Multiple Power Supply Library Requirements
- Modeling Libraries With Integrated Clock-Gating Cells
- Modeling Electromigration

Modeling Power Terminology

The power a circuit dissipates falls into two broad categories:

- Static power
 - Dynamic power
-

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. This leakage is caused by reduced threshold voltages that prevent the gate from turning off completely. Static power also results when current leaks between the diffusion layers and substrate. For this reason, static power is often called *leakage power*.

Dynamic Power

Dynamic power is the power dissipated when a 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 a net can change without necessarily resulting in a logic transition, dynamic power can result even when a net does not change its logic state.

The dynamic power of a circuit is composed of

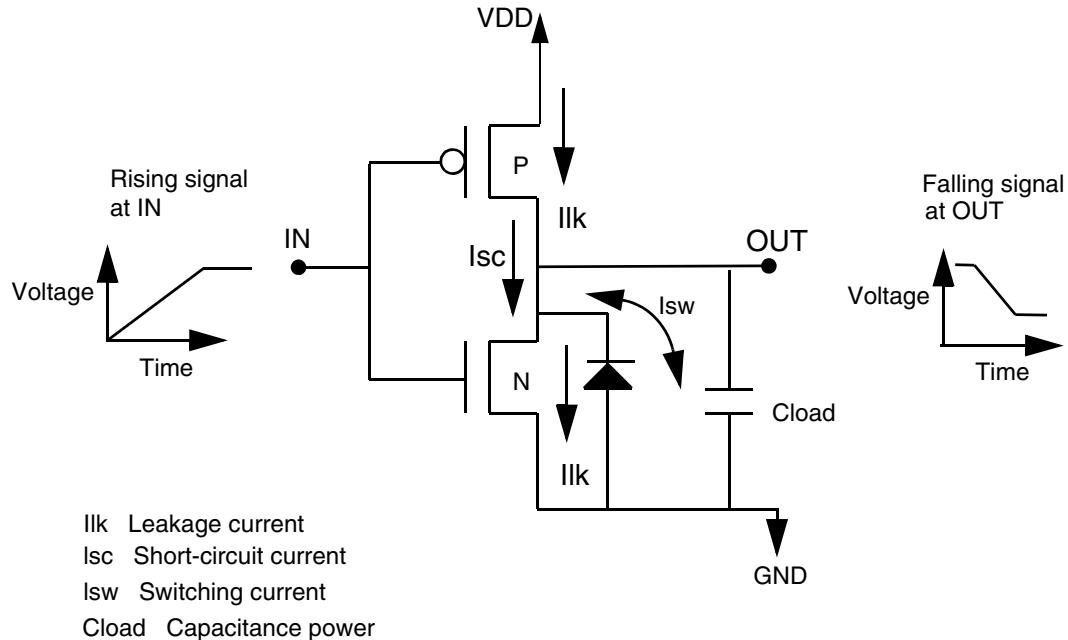
- Internal power
- Switching power

Internal Power

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

[Figure 12-1](#) illustrates components of power dissipation and shows the cause of short-circuit power. In this figure, there is a slow rising signal at the gate input IN. As the signal makes a transition from low to high, the N-type transistor turns on and the P-type transistor turns off. However, during signal transition, both the P- and N-type transistors can be on simultaneously for a short time. During this time, current flows from VDD to GND, resulting in short-circuit power.

Figure 12-1 Components of Power Dissipation



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

In most simple library cells, internal power is due primarily to short-circuit power. For this reason, the terms *internal power* and *short-circuit power* are often considered synonymous.

Note:

A transition implies either a rising or a falling signal; therefore, if the power characterization involves running a full-cycle simulation, which includes both rising and falling signals, then you must average the energy dissipation measurement by dividing by 2.

Switching Power

The switching power, or capacitance 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 driver.

The equation for switching power is

$$(1/2)CV^2 \cdot TR_i$$

where TR_i is the toggle-rate activity. This equation is applied to each net in the design.

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

[Figure 12-1](#) shows how the capacitance (C_{load}) is charged and discharged as the N or P transistor turns on. Switching power accounts for 70 to 90 percent of the power dissipation of an active CMOS circuit.

Note:

Two measures of power dissipation are useful to designers: average power and peak power. Average power is the power dissipated by a circuit over a representative set of input stimuli. Peak power is the maximum power dissipated by a circuit over all the input stimuli. Peak power is normally associated with a particular stimulus.

Switching Activity

Switching activity is a metric used to measure the number of transitions (0-to-1 and 1-to-0) for every net in a circuit when input stimuli are applied. Switching activity is the average activity of the circuit with a set of input stimuli.

A circuit with higher switching activity is likely to dissipate more power than a circuit with lower switching activity. Switching activity is also correlated with the random-pattern testability of the circuit. A circuit with higher switching activity implies that a randomly selected input pattern might have better coverage.

For more information about switching activity and power consumption, see the *Power Compiler User Guide*.

Modeling Cells for Power

The three components of power dissipation are

- Leakage power
- Internal (short-circuit) power
- Switching power

As this equation shows, leakage power is summed over all cells in the design to yield the design's total leakage power (total static power dissipation):

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

$P_{\text{LeakageTotal}}$

Total leakage power dissipation of the design.

$P_{\text{CellLeakage}}$

Leakage power dissipation of the cell.

As the following equation shows, the internal power of the cells and the switching power of the nets are used to compute the design's total dynamic power dissipation:

$$P_{\text{Dynamic}} = \sum_{\forall \text{cells}(i)} (P_{\text{Cellinternal}_i} \times E_i) + \frac{V_{dd}^2}{2} \sum_{\forall \text{nets}(i)} (C_{\text{Load}_i} \times E_i)$$

[Internal power] [Switching power]

P_{Dynamic}

Total dynamic power dissipation of the design.

$P_{\text{Cellinternal}}$

Internal power of each cell.

V_{dd}

Supply voltage.

C_{Load}

Capacitive load of each net.

The unit for switching power and internal power is a derived unit. It is derived from the following function:

$$(capacitive_load_unit \cdot voltage_unit^2) / time_unit$$

For more information about dynamic power unit derivation, see the *Power Compiler User Guide*.

Modeling for Leakage Power

Regardless of the physical reasons for leakage power, library developers can annotate gates with the approximate total leakage power dissipated by the gate.

Leakage power characterization requires measuring the supply current (I_{supply}) of the quiescent state of the cell. This requires a DC analysis of the circuit with steady-state voltages at the inputs to the cell.

Vendors can characterize leakage power of multiple combinations of input states to generate state-dependent leakage power models. This is especially important when leakage power dissipation varies greatly from one state to another and requires iterating across all possible inputs and measuring the supply current for each vector. For each vector or state, leakage power can be computed as

$$P_{\text{leakage}} = VDD \times I_{\text{supply}}$$

Alternatively, leakage power can also be characterized with two simulation runs: one for output high and one for output low. The average of these two measurements is then used as the cell's leakage power. This method reduces the characterization effort at the expense of accuracy.

Representing Leakage Power Information

You can represent leakage power information in the library as:

- Cell-level state-independent leakage power with the `cell_leakage_power` attribute
- Cell-level state-dependent leakage power with the `leakage_power` group
- The associated library-level attributes that specify scaling factors, units, and a default for both leakage and power density

cell_leakage_power Simple Attribute

This attribute specifies the leakage power of a cell. If this attribute is missing or negative, the value of the `default_cell_leakage_power` attribute is used.

Syntax

```
cell_leakage_power : value ;
```

Note:

You must define this attribute for cells with state-dependent leakage power to provide the leakage power value for those states where the state-specific leakage power has not been specified using the `leakage_power` group. Otherwise, the Library Compiler tool issues an error.

Using the `leakage_power` Group for a Single Value

This group specifies the leakage power of a cell when the leakage power depends on the logical condition of that cell. This type of leakage power is called state-dependent. To model state-dependent leakage power, use the following attributes:

- mode
- when
- value

Syntax

```
leakage_power ( ) {
    mode (mode_name, mode_value);
    when : "Boolean expression";
    value: float;
}
```

mode Complex Attribute

You define the `mode` attribute within a `leakage_power` group. A `mode` attribute pertains to an individual cell. The cell is active when `mode` is instantiated with a name and a value. You can specify multiple instances of the `mode` attribute but only one instance for each cell.

Syntax

```
mode (mode_definition_name, mode_value);
```

The Library Compiler tool issues an error message if the `mode_name` and `mode_value` strings are not already defined by a `mode_definition` group in the cell.

Example

```
cell (my_cell) {
    mode_definition (mode_definition_name) {
        mode_value(namestring) {
            when : "Boolean expression";
            sdf_cond : "Boolean expression";
        } ...
    ...
    leakage_power (namestring) {
        mode (mode_name, mode_value);
```

```

/*when : "Boolean expression";*/
value : float;
...
}/* end leakage_power() */
leakage_current() { /* without the when statement */
/* default state */
...
}
}/* end pin B */
}/* end cell */

```

[Example 12-1](#) shows a mode instance description.

Example 12-1 A mode Instance Description

```

library (my_library) {
  technology ( cmos ) ;
  delay_model : table_lookup;
  ...
  cell(inv0d0) {
    area : 0.75;
    mode_definition(rw) {

      mode_value(read) {
        when : "I";
        sdf_cond : "I == 1";
      }
      mode_value(write) {
        when : "!I";
        sdf_cond : "I == 0";
      }
    }
    leakage_power () {
      mode(rw, read);
      value : 2;
    }
    leakage_power () {
      mode(rw, write);
      value : 2.2;
    }
    pin(I) {
      direction : input;
      max_transition : 2100.0;
      capacitance : 0.002000;
      fanout_load : 1;
      ...
    }
    ...
  } /* cell(inv0d0) */
} /* library

```

when Simple Attribute

This attribute specifies the state-dependent condition that determines whether the leakage power is accessed.

Syntax

```
when : "Boolean expression";
```

Boolean expression

The name or names of pins, buses, and bundles with their corresponding Boolean operators. [Table 12-1](#) lists valid operators.

Table 12-1 Valid Boolean Operators

Operator	Description
,	invert previous expression
!	invert following expression
^	logical XOR
*	logical AND
&	logical AND
space	logical AND
+	logical OR
	logical OR
1	signal tied to logic 1
0	signal tied to logic 0

The order of precedence of the operators is left to right, with inversion performed first, then XOR, then AND, then OR.

You must define mutually exclusive conditions for state-dependent leakage power and internal power. Mutually exclusive means that only one condition defined in the `when` attribute can be met at any given time. You can reference buses and bundles in the `when` attribute in the `leakage_power` group, as shown here:

Syntax

```
when : "bus_name";
```

Example

```
cell(Z) {
  ...
  leakage_power () {
```

```

    when : "A";
    ...
}
}

```

value Simple Attribute

The value attribute represents the leakage power for a given state of a cell. The value for this attribute is a floating-point number.

Example

```

cell (my cell) {
    ...
    leakage_power () {
        when : "! A";
        value : 2.0;
    }
    cell_leakage_power : 3.0;
}

```

leakage_power_unit Simple Attribute

This attribute indicates the units of the leakage-power values in the library. [Table 12-2](#) shows the possible unit values you can enter and their corresponding mathematical symbol equivalent.

Table 12-2 Valid Unit Values and Mathematical Equivalents

Text entry	Mathematical equivalent
1mW	1mW
100uW	100 micro W
10uW	10 micro W
1uW	1 micro W
100nW	100nW
10nW	10nW
1nW	1nW
100pW	100pW

Table 12-2 Valid Unit Values and Mathematical Equivalents (Continued)

Text entry	Mathematical equivalent
10pW	10pW
1pW	1pW

Example

```
leakage_power_unit : 100uW;
```

If this attribute is missing, the leakage-power values are expressed without units.

default_cell_leakage_power Simple Attribute

The `default_cell_leakage_power` attribute indicates the default leakage power for those cells for which you have not set the `cell_leakage_power` attribute. This attribute must be a nonnegative floating-point number. The default is 0.0.

Example

```
default_cell_leakage_power : 0.5;
```

The Library Compiler tool issues a warning if the library has `cell_leakage_power` information but does not have the `default_cell_leakage_power` attribute defined.

Example

```
library(leakage) {
    delay_model : table_lookup;

    /* unit attributes */
    time_unit : "1ns";
    voltage_unit : "1V";
    current_unit : "1mA";
    pulling_resistance_unit : "1kohm";
    leakage_power_unit : "1pW";
    capacitive_load_unit (1.0,pf);

    cell (NAND2) {

        cell_leakage_power : 1.0 ;
        leakage_power() {
            related_pg_pin : "VDD1";
            when : "!A1 !A2" ;
```

```
    value : 1.5 ;
}
leakage_power() {
    related_pg_pin : "VDD1";
    when : "!A1 A2" ;
    value : 2.0 ;
}
leakage_power() {
    related_pg_pin : "VDD1";
    when : "A1 !A2" ;
    value : 3.0 ;
}
leakage_power() {
    related_pg_pin : "VDD1";
    when : "A1 A2" ;
    value : 4.0 ;
}
leakage_power() {
    related_pg_pin : "VDD2";
    when : "!A1 !A2" ;
    value : 3.5 ;
}
leakage_power() {
    related_pg_pin : "VDD2";
    when : "!A1 A2" ;
    value : 3.0 ;
}
leakage_power() {
    related_pg_pin : "VDD2";
    when : "A1 !A2" ;
    value : 4.0 ;
}
leakage_power() {
    related_pg_pin : "VDD2";
    when : "A1 A2" ;
    value : 5.0 ;
}
area : 1.0 ;
pin(A1) {
    direction : input;
    capacitance : 0.1 ;
}
pin(A2) {
    direction : input;
    capacitance : 0.1 ;
}
pin(ZN) {
    direction : output;
    max_capacitance : 0.1;
    function : "(A1*A2)'";
    timing() {
        timing_sense : "negative_unate"
        related_pin : "A1"
```

```
cell_rise( scalar ) {
    values("0.0");
}
rise_transition( scalar ) {
    values("0.0");
}
cell_fall( scalar ) {
    values("0.0");
}
fall_transition( scalar ) {
    values("0.0");
}
internal_power() {
    related_pin : "A1"
    related_pg_pin : "VDD1";
    rise_power( scalar ) {
        values("0.0");
    }
    fall_power( scalar ) {
        values("0.0");
    }
} /* end of internal power */
internal_power() {
    related_pin : "A1"
    related_pg_pin : "VDD2";
    rise_power( scalar ) {
        values("0.0");
    }
    fall_power( scalar ) {
        values("0.0");
    }
} /* end of internal power */
} /* end of timing for related pin A1 */
timing() {
    timing_sense : "negative_unate"
    related_pin : "A2"
    cell_rise( scalar ) {
        values("0.0");
    }
    rise_transition( scalar ) {
        values("0.0");
    }
    cell_fall( scalar ) {
        values("0.0");
    }
    fall_transition( scalar ) {
        values("0.0");
    }
    internal_power() {
        related_pin : "A2"
        related_pg_pin : "VDD1";
        rise_power( scalar ) {
            values("0.0");
        }
    }
}
```

```

    }
    fall_power( scalar ) {
        values("0.0");
    }
} /* end of internal power */
internal_power() {
    related_pin : "A2"
    related_pg_pin : "VDD2";
    rise_power( scalar ) {
        values("0.0");
    }
    fall_power( scalar ) {
        values("0.0");
    }
} /* end of internal power */
} /* end of timing for related pin A2 */
} /* end of pin ZN */
} /* end of cell */
} /* end of library */

```

Threshold Voltage Modeling

Multiple threshold power saving flows require library cells to be categorized according to the transistor's threshold voltage characteristics, such as high threshold voltage cells and low threshold voltage cells. High threshold voltage cells exhibit lower power leakage but run slower than low threshold voltage cells.

You can specify low threshold voltage cells and high threshold voltage cells in the same library, simplifying library management and setup, by using the following optional attributes:

- `default_threshold_voltage_group`

Specify the `default_threshold_voltage_group` attribute at the library level, as shown, to specify a cell's category based on its threshold voltage characteristics:

```
default_threshold_voltage_group : group_nameid ;
```

The `group_name` value is a string representing the name of the category, such as `high_vt_cell` to represent a high voltage cell.

- `threshold_voltage_group`

Specify the `threshold_voltage_group` attribute at the cell level, as shown, to specify a cell's category based on its threshold voltage characteristics:

```
threshold_voltage_group : group_nameid ;
```

The `group_name` value is a string representing the name of the category. Typically, you would specify a pair of `threshold_voltage_group` attributes, one representing the high voltage and one representing the low voltage. However, there is no limit to the number of

`threshold_voltage_group` attributes you can have in a library. If you omit this attribute, the value of the `default_threshold_voltage_group` attribute is applied to the cell.

Example

```
library ( mixed_vt_lib ) {
...
default_threshold_voltage_group : "high_vt_cell" ;
...
cell(ht_cell) {
    threshold_voltage_group : "high_vt_cell" ;
    ...
}
cell(lt_cell) {
    threshold_voltage_group : "low_vt_cell" ;
    ...
}
```

Modeling for Internal and Switching Power

These are two compatible definitions of internal or short-circuit power:

- Short-circuit power is the power dissipated by the instantaneous short-circuit connection between Vdd and GND while the gate is in transition.
- Internal power is all the power dissipated within the boundary of the gate. This definition does not distinguish between the cell's short-circuit power and the component of switching power that is being dissipated internally to the cell as a result of the drain-to-substrate capacitance that is being charged and discharged. In this definition, the interconnect switching power is the power dissipated because of lumped wire capacitance and input pin capacitances but not because of the output pin capacitance.

Library developers must choose one of these definitions and specify internal power and capacitance numbers accordingly. Library developers can choose

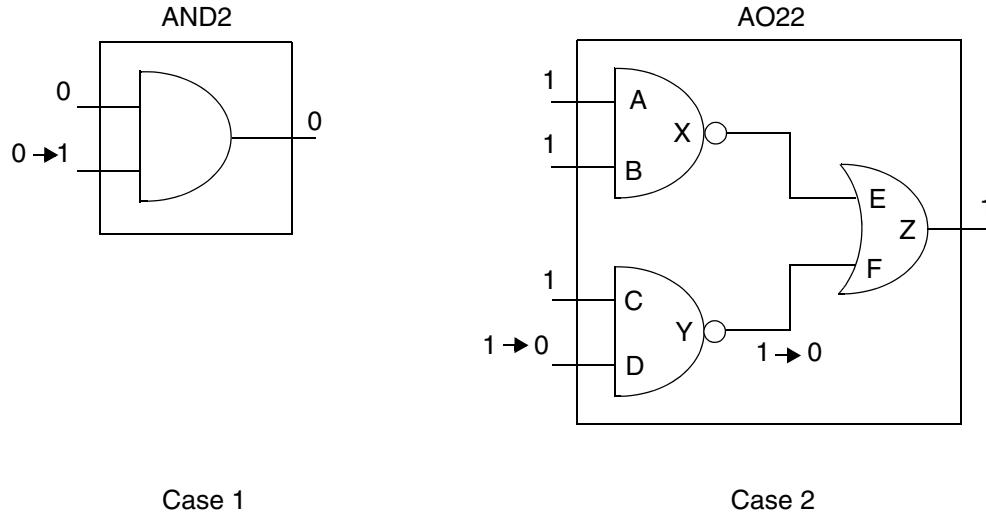
- To include the effect of the output capacitance in the `internal_power` attribute, which gives the output pins zero capacitance
- To give the output pins a real capacitance, which causes them to be included in the switching power, and model only short-circuit power as the cell's internal power

Together, internal power and switching power contribute to the total dynamic power dissipation. Like switching power, internal power is dissipated whenever an output pin makes a transition.

Some power is also dissipated as a result of internal transitions that do not cause output transitions. However, those are relatively minor in comparison (they consume much less power) and should be ignored.

[Figure 12-2](#) shows two examples of an input transition that does not cause a corresponding output transition.

Figure 12-2 Complex Gate Example



In Case 1, input B of the AND2 gate undergoes a 0-to-1 transition but the output remains stable at 0. This might consume a small amount of power as one of the N-transistors opens, but the current flow is very small.

In Case 2, input D of the multilevel gate AO22 undergoes a 1-to-0 transition, causing a 1-to-0 transition at internal pin Y. However, output Z remains stable at 1. The significance of the power dissipation in this case depends on the load of the internal wire connected to Y. In Case 1, power dissipation is negligible, but in Case 2, power dissipation might result in some inaccuracy.

You can set the `internal_power` group attribute so that multiple input or output pins that share logic can transition together within the same time period.

Pins transitioning within the same time period can lower the level of power consumption.

Modeling Internal Power Lookup Tables

You should measure the energy dissipated by varying either input voltage transition or output load while holding the other constant. Because a table indexed by T input transition times and C output load capacitances has $T \times C$ entries, the cell's internal power must be

characterized TxC times, one time for each input transition time and output load capacitance combination. For example, if internal power is modeled by use of a 3x3 table at the output of the cell, the design has 9 input voltage transitions—output load combinations where energy dissipation must be measured.

The `library` group supports a one-, two-, or three-dimensional `internal_power` lookup table indexed by the total output load capacitances (best model), the input transition time, or both. The internal power lookup table uses the same syntax as the nonlinear lookup table for delay.

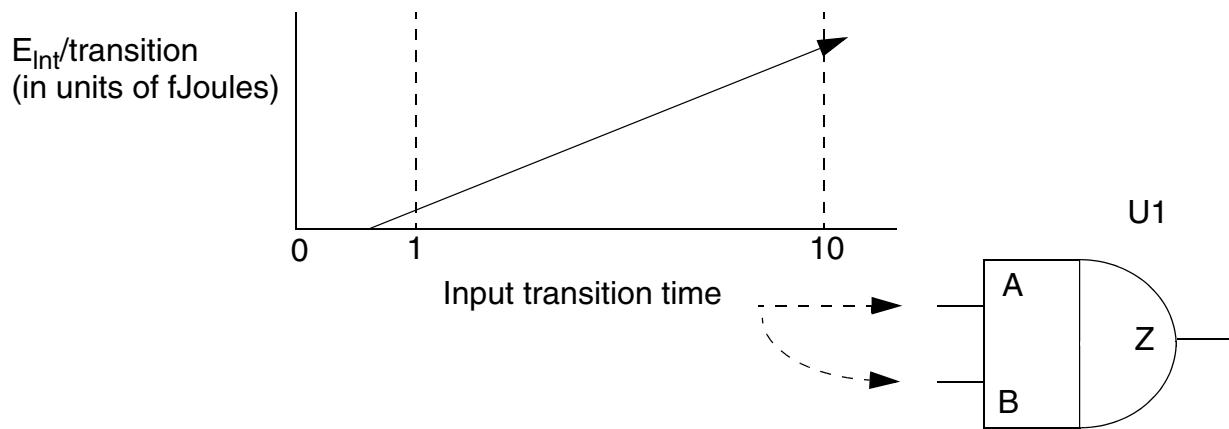
You can set the `internal_power` attributes for input pins, which are indexed by input transition time. In this way, you can model gate AO22 in [Figure 12-2](#) or, more important, the power consumed by the flip-flop clock or reset pins.

Note:

The input pin power is added to the output pin power. When you model the library, avoid double counting.

[Figure 12-3](#) shows how to calculate the input pin power information for the one-dimensional lookup table describing internal cell U1.

Figure 12-3 Internal Power for Cell U1



To calculate the internal power for cell U1, use the following equation:

$$P_{Int} = (E_Z \times AF_Z) + (E_A \times AF_A) + (E_B \times AF_B)$$

P_{Int}

Total internal power for the cell.

E

Internal energy for the pin.

AF

Activity factor.

Accurate sequential modeling requires a separate table for the clock and for the output pin the clock controls. The two tables are used to ensure that clock pin power and output power are accounted for separately, because a clock pin often toggles without causing any observable state change on the output pin. The separate power table scheme ensures that power dissipated within the cell is accounted for properly when the clock pin toggles but the output pin does not. The following discussion pertains to single-output, single-clock sequential cells, but the concept is also extendable to multioutput, multiclock cells.

Clock Pin Power

This energy is characterized by simulation of a single full cycle (one rise transition and one fall transition) of the clock, with no transition at the output and input pins. A one-dimensional internal power table indexed by input transition time should be attached to the clock pin. Total energy dissipated in the cell during this simulation is measured. If separate rise and fall power modeling is not used, the energy measured must be divided by 2 to get the energy dissipated by the clock pin transition, because the measurement is done for two transitions of the clock.

```
Clk_Pin_Energy = Clk_Total / 2
```

Add `Clk_Pin_Energy` as an entry indexed by input transition time in the one-dimensional internal power table attached to the clock pin.

Output Pin Power

This power is characterized by simulation of two full cycles of the clock, with two rise and fall transitions at the output. A two-dimensional internal power table should be attached to the output pin. Total energy dissipated in the cell during the two-full-cycle simulation (`Out_total`) is measured. If separate rise and fall power modeling is not used, the energy measured must be divided by 2, because the measurement is done for two transitions.

```
Output_Pin_Energy = (Out_total)/2 - 2*(Clk_Pin_Energy)
```

or

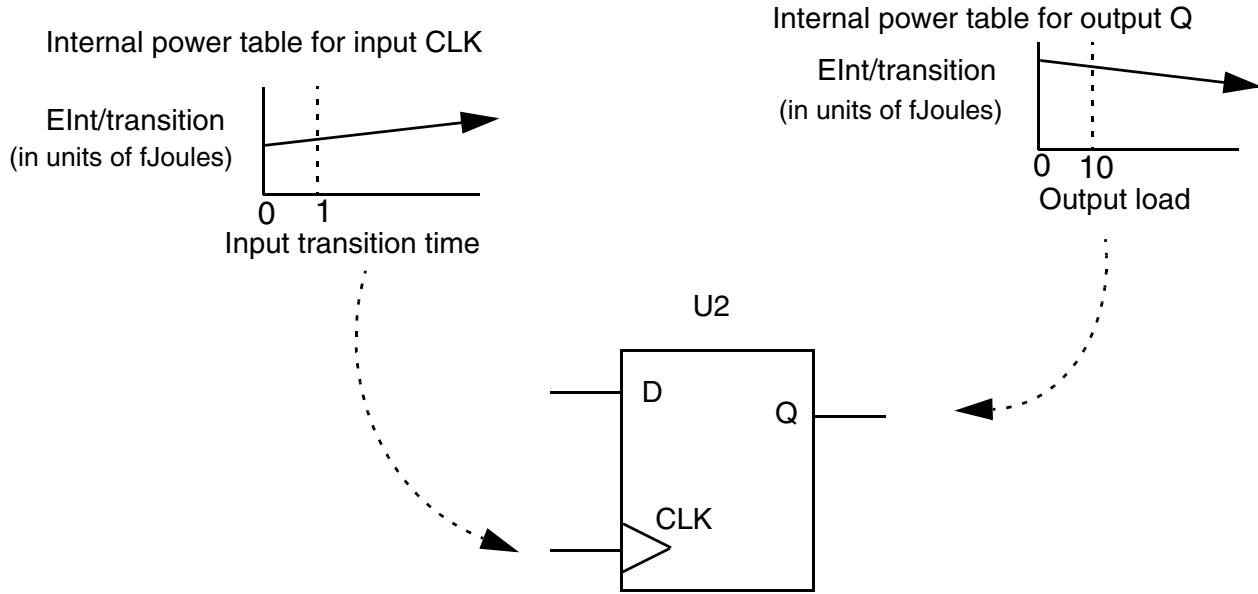
```
Output_Pin_Energy = (Out_total)/2 - Clk_Total
```

Note:

Note that the output pin energy is adjusted by subtraction of the input pin power. This prevents double counting during simulation.

[Figure 12-4](#) shows how to calculate the input pin power information for internal cell U2.

Figure 12-4 Internal Power for Cell U2



To calculate the internal power for cell U2, use the following equation:

$$P_{\text{Int}} = (E_Q \times AF_Q) + (E_{\text{CLK}} \times AF_{\text{CLK}})$$

P_{Int}

Total internal power for the cell.

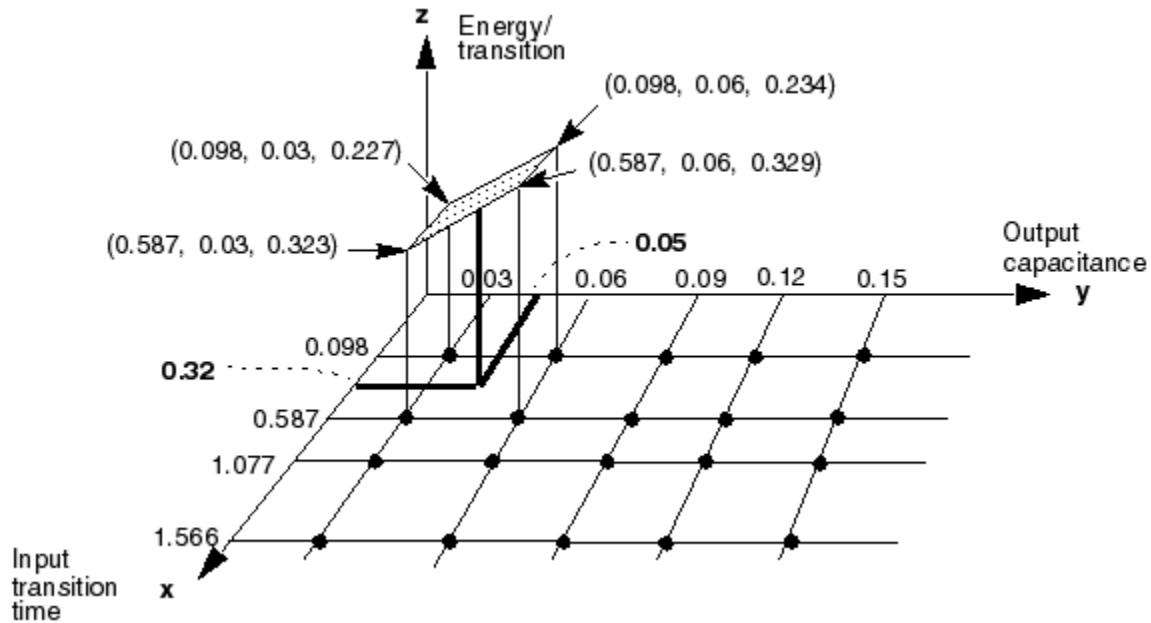
E

Internal energy for the pin.

AF

Activity factor.

[Figure 12-5](#) is an example of the two-dimensional lookup table for modeling output pin power in a cell.

Figure 12-5 Internal Power Table for Cell Output

Calculating Switching Power

Switching (or interconnect) power is the power dissipated by the capacitive load on a net whenever the net makes a logical transition. Power is dissipated when the capacitive load is charged or discharged. With internal power, switching power is used to compute the design's total dynamic power dissipation.

Switching power information is a function of a net's toggle rate, capacitive loading, associated clock frequency, and the supply voltage level of the design. The Synopsys library model supports all of these parameters except for the toggle rate.

An explicit units attribute is not required for switching power, because the units are implicitly determined by the units of the voltage, time, and capacitance attributes.

Because all toggle rates are internally adjusted to a period defined by the library-level `time_unit` attribute, the resulting value of switching power is defined in terms of joules per second (or watts) multiplied by the appropriate power of 10, as determined by the units for time, capacitance, and voltage. For example, in a library with time units of 1 ns, capacitive units of 0.1 femtofarad, and voltage units of 1 volt, the calculation for the derived units for the library's switching power is:

$$\text{Power_Units} = \frac{(1 \text{ V}^2) \times 0.1 \text{ ff}}{1 \text{ ns}} = 0.1 \mu\text{W}$$

For a single net with a total load of 100 femtofarad, a toggle rate of two transitions every 100 ns, and a supply voltage of 5 volts, the calculation of the net's power dissipation is:

$$\begin{aligned}\text{Net_Power} &= \frac{V_{dd}^2}{2} \sum_{\forall \text{nets}(i)} (C_{Load_i} \times TR_i) \\ &= \frac{5^2}{2} \times 100 \times \frac{2}{100} \times (0.1 \mu W) \\ &= 25 \times (0.1 \mu W) \\ &= 2.5 \mu W\end{aligned}$$

TR

Toggle rate (number of toggles per unit of time).

C_{Load}

Capacitive load of each net.

Representing Internal Power Information

You can describe power dissipation in your libraries by using lookup tables.

Specifying the Power Model

Use the library level `power_model` attribute to specify the power model for your library. The valid value is `table_lookup`. If you do not specify a power model, the Library Compiler tool assumes `table_lookup`.

Using Lookup Table Templates

To represent internal power, you can create templates of common information that multiple lookup tables can use. Use the following groups and attributes to define your lookup tables:

- The library-level `power_lut_template` group
- The `internal_power` group (see “[Defining Internal Power Groups](#)” on page 12-24)
- The associated library-level attributes that specify the scaling factors and a default

power_lut_template Group

Use this library-level group to create templates of common information that multiple lookup tables can use. A table template specifies the table parameters and the breakpoints for each axis. Assign each template a name. Make the template name the group name of a `fall_power` group, `power` group, or `rise_power` group in the `internal_power` group; doing so enables the power lookup tables to refer to the template.

Syntax

```
power_lut_template(name) {
    variable_1 : string ;
    variable_2 : string ;
    variable_3 : string ;
    index_1("float, ... , float") ;
    index_2("float, ... , float") ;
    index_3("float, ... , float") ;
}
```

Template Variables

The lookup table template for specifying power uses three associated variables to characterize cells in the library for internal power:

- `variable_1`, which specifies the first dimensional variable
- `variable_2`, which specifies the second dimensional variable
- `variable_3`, which specifies the third dimensional variable

These variables indicate the parameters used to index into the lookup table along the first, second, and third table axes.

Following are valid values for `variable_1`, `variable_2`, and `variable_3`:

`total_output_net_capacitance`

The loading of the output net capacitance of the pin specified in the `pin` group that contains the `internal_power` group.

`equal_or_opposite_output_net_capacitance`

The loading of the output net capacitance of the pin specified in the `equal_or_opposite_output` attribute in the `internal_power` group.

`input_transition_time`

The input transition time of the pin specified in the `related_pin` attribute in the `internal_power` group.

For information about the `related_pin` attribute, see “[Defining Internal Power Groups](#)” on page 12-24.

Template Breakpoints

The index statements define the breakpoints for an axis. The breakpoints defined by `index_1` correspond to the parameter values indicated by `variable_1`. The breakpoints defined by `index_2` correspond to the parameter values indicated by `variable_2`. The breakpoints defined by `index_3` correspond to the parameter values indicated by `variable_3`.

You can overwrite the `index_1`, `index_2`, and `index_3` attribute values by providing the same `index_x` attributes in the `fall_power` group, `power` group, or `rise_power` group in the `internal_power` group.

The index values are lists of floating-point numbers greater than or equal to 0.0. The values in the list must be in an increasing order.

The number of floating-point numbers in the indexes determines the size of each dimension, as [Example 12-2](#) illustrates.

For each `power_lut_template` group, you must define at least one `variable_1` and one `index_1`.

[Example 12-2](#) shows four `power_lut_template` groups that have one-, two-, or three-dimensional templates.

Example 12-2 Four power_lut_template Groups

```
power_lut_template (output_by_cap) {
    variable_1 : total_output_net_capacitance ;
    index_1 ("0.0, 5.0, 20.0") ;
}

power_lut_template (output_by_cap_and_trans) {
    variable_1 : total_output_net_capacitance ;
    variable_2 : input_transition_time ;
    index_1 ("0.0, 5.0, 20.0") ;
    index_2 ("0.1, 1.0, 5.0") ;
}

power_lut_template (input_by_trans) {
    variable_1 : input_transition_time ;
    index_1 ("0.0, 1.0, 5.0") ;
}

power_lut_template (output_by_cap2_and_trans) {
    variable_1 : total_output_net_capacitance ;
    variable_2 : input_transition_time ;
    variable_3 : equal_or_output_net_capacitance ;
    index_1 ("0.0, 5.0, 20.0") ;
    index_2 ("0.1, 1.0, 5.0") ;
    index_3 ("0.1, 0.5, 1.0") ;
}
```

Scalar power_lut_template Group

Use this group to model cells with no power consumption.

The syntax has a predefined template named scalar; its value size is 1. You can specify scalar as the group name of a `fall_power` group, `power` group, or `rise_power` group in the `internal_power` group.

Example

```
internal_power() {  
    power(scalar) {  
        values ("0.0");  
    }  
}
```

Note:

You can use the scalar template with an assigned value of 0 (indicating that no power is consumed) for an `internal_power` group with a `rise_power` table and a `fall_power` table. When one group contains the scalar template, the other must contain one-, two-, or three-dimensional power values.

Defining Internal Power Groups

To specify the cell's internal power consumption, use the `internal_power` group within the `pin` group in the cell.

If the `internal_power` group is not present in a cell, it is assumed that the cell does not consume any internal power. You can define the optional complex attribute `index_1`, `index_2`, or `index_3` in this group to overwrite the `index_1`, `index_2`, or `index_3` attribute defined in the library-level `power_lut_template` to which it refers (see [Example 12-5 on page 12-44](#)).

Naming Power Relationships, Using the `internal_power` Group

Within the `internal_power` group you can identify the name or names of different power relationships. A single power relationship can occur between an identified pin and a single related pin identified with the `related_pin` attribute. Multiple power relationships can occur in many ways.

This list shows seven possible multiple power relationships. These relationships are described in more detail in the following sections:

- Between a single pin and a single related pin
- Between a single pin and multiple related pins

- Between a bundle and a single related pin
- Between a bundle and multiple related pins
- Between a bus and a single related pin
- Between a bus and multiple related pins
- Between a bus and related bus pins

Power Relationship Between a Single Pin and a Single Related Pin

Identify the power relationship that occurs between a single pin and a single related pin by entering a name in the `internal_power` group attribute as shown in the following example:

Example

```
cell (my_inverter) {
    ...
    pin (A) {
        direction : input;
        capacitance : 1;
    }
    pin (B) {
        direction : output
        function : "A'";
        internal_power (A_B) {
            related_pin : "A";
        }
        ...
    }/* end internal_power() */
}/* end pin B */
}/* end cell */
```

The power relationship is as follows:

From pin	To pin	Label
A	B	A_B

Power Relationships Between a Single Pin and Multiple Related Pins

This section describes how to identify the power relationships when an `internal_power` group is within a `pin` group and the power relationship has more than a single related pin. You identify the multiple power relationships on a name list entered with the `internal_power` group attribute as shown in the following example:

Example

```
cell (my_and) {
```

```

...
pin (A) {
    direction : input;
    capacitance : 1;
}
pin (B) {
    direction : input;
    capacitance : 2;
}
pin (C) {
    direction : output
    function : "A B";
    internal_power (A_C, B_C) {
        related_pin : "A B";
    }
}
/* end internal_power() */
/* end pin B */
/* end cell */

```

The power relationships are as follows:

From pin	To pin	Label
A	C	A_C
B	C	B_C

Power Relationships Between a Bundle and a Single Related Pin

When the `internal_power` group is within a `bundle` group that has several members that have a single related pin, enter the names of the resulting multiple power relationships in a name list in the `internal_power` group.

The Library Compiler tool assumes that the first name in the name list is the relationship between the related pin and the first pin in the bundle member list, the second name in the name list is the relationship between the related pin and the second pin in the bundle member list, and so on.

Example

```

...
bundle (Q) {
    members (Q0, Q1, Q2, Q3);
    direction : output;
    function : "IQ";
    internal_power (G_Q0, G_Q1, G_Q2, G_Q3) {
        related_pin : "G";
    }
}

```

If G is a pin, as opposed to another `bundle` group, the power relationships are as follows:

From pin	To pin	Label
G	Q0	G_Q0
G	Q1	G_Q1
G	Q2	G_Q2
G	Q3	G_Q3

If G is another bundle of member size 4 and G0, G1, G2, and G3 are members of bundle G, the power relationships are as follows:

From pin	To pin	Label
G0	Q0	G_Q0
G1	Q1	G_Q1
G2	Q2	G_Q2
G3	Q3	G_Q3

Note:

If G is a bundle of a member size other than 4, it is an error due to incompatible width.

Power Relationships Between a Bundle and Multiple Related Pins

When the `internal_power` group is within a `bundle` group that has several members, each having a corresponding related pin, enter the names of the resulting multiple power relationships as a name list in the `internal_power` group.

The Library Compiler tool assumes that the first name in the name list is the relationship between the related pin and the first pin in the bundle member list, the second name in the name list is the relationship between the second related pin and the second pin in the bundle member list, and so on.

Example

```
bundle (Q) {
    members (Q0, Q1, Q2, Q3);
    direction : output;
    function : "IQ";
    internal_power (G_Q0, H_Q0, G_Q1, H_Q1, G_Q2, H_Q2, G_Q3, H_Q3) {
        related_pin : "G H";
    }
}
```

}

If G is a pin, as opposed to another `bundle` group, the power relationships are as follows:

From pin	To pin	Label
G	Q0	G_Q0
H	Q0	H_Q0
G	Q1	G_Q1
From pin	To pin	Label
H	Q1	H_Q1
G	Q2	G_Q2
H	Q2	H_Q2
G	Q3	G_Q3
H	Q3	H_Q3

If G is another bundle of member size 4 and G0, G1, G2, and G3 are members of bundle G, the power relationships are as follows:

From pin	To pin	Label
G0	Q0	G_Q0
H	Q0	H_Q0
G1	Q1	G_Q1
H	Q1	H_Q1
G2	Q2	G_Q2
H	Q2	H_Q2
G3	Q3	G_Q3
H	Q3	H_Q3

The same rule applies if H is a size 4 bundle.

Note:

If G is a bundle of a member size other than 4, it's an error due to incompatible width.

Power Relationships Between a Bus and a Single Related Pin

This section describes how to identify the power relationships created when an `internal_power` group is within a `bus` group that has several bits that have the same single related pin. You identify the resulting multiple power relationships by entering a name list, using the `internal_power` attribute.

The Library Compiler tool assumes that the first name in the name list identifies the relationship between the related pin and the most significant bit (MSB) in the `bus` group, the second name in the name list identifies the relationship between the related pin and the second MSB in the `bus` group, and so on.

Example

```
...
bus (X) {
    /*assuming MSB is X[0] */
    bus_type : bus4;
    direction : output;
    capacitance : 1;
    pin (X[0:3]) {
        function : "B'";
        internal_power (B_X0, B_X1, B_X2, B_X3) {
            related_pin : "B";
        }
    }
}
```

If B is a pin, as opposed to another 4-bit bus, the power relationships are as follows:

From pin	To pin	Label
B	X[0]	B_X0
B	X[1]	B_X1
B	X[2]	B_X2
B	X[3]	B_X3

If B is another 4-bit bus and B[0] is the MSB for bus B, the power relationships are as follows:

From pin	To pin	Label
B[0]	X[0]	B_X0
B[1]	X[1]	B_X1
B[2]	X[2]	B_X2
B[3]	X[3]	B_X3

Power Relationships Between a Bus and Multiple Related Pins

This section describes the power relationships created when an `internal_power` group is within a `bus` group that has several bits, where each bit has its own related pin. You identify the resulting multiple power relationships by entering a name list, using the `internal_power` group attribute.

The Library Compiler tool assumes that the first name in the name list is the relationship between the first related pin and the most significant bit (MSB) in the `bus` group, the second name in the name list is the relationship between the second related pin and the second MSB in the `bus` group, and so on.

Example

```
bus (X) { /*assuming MSB is X[0] */
    bus_type : bus4;
    direction : output;
    capacitance : 1;
    pin (X[0:3]) {
        function : "B'";
        internal_power (B_X0, C_X0, B_X1, C_X1, B_X2, C_X2, B_X3,C_X3) {
            related_pin : "B' C";
        }
    }
}
```

If B and C are pins, as opposed to another 4-bit bus, the power relationships are as follows:

From pin	To pin	Label
B	X[0]	B_X0
C	X[0]	C_X0
B	X[1]	B_X1
C	X[1]	C_X1
B	X[2]	B_X2
C	X[2]	C_X2
B	X[3]	B_X3
C	X[3]	C_X3

If B is another 4-bit bus and B[0] is the MSB for bus B, the power relationships are as follows:

From pin	To pin	Label
B[0]	X[0]	B_X0
C	X[0]	C_X0
B[1]	X[1]	B_X1
C	X[1]	C_X1
B[2]	X[2]	B_X2
C	X[2]	C_X2
B[3]	X[3]	B_X3
C	X[3]	C_X3

The same rule applies if C is a 4-bit bus.

Power Relationships Between a Bus and Related Bus Pins

This section describes the power relationships created when an `internal_power` group is within a `bus` group that has several bits that have to be matched with several related bus pins of a required width. Identify the resulting multiple power relationships by entering a name list, using the `internal_power` group attribute.

The first name in the name list is the relationship between the first pin of the related bus pins of the necessary width and the MSB in the `bus` group, the second name is the relationship between the second pin of the related bus pins and the second MSB in the `bus` group, and so on.

Example

```
/* assuming related_bus_pins is width of 2 bits */
bus (X) {
    /*assuming MSB is X[0] */
    bus_type : bus4;
    direction : output;
    capacitance : 1;
    pin (X[0:3]) {
        function : "B'";
        internal_power (B0_X0, B0_X1, B0_X2, B0_X3, B1_X0, B1_X1, \
                        B1_X2,B1_X3) {
            related_bus_pins : "B";
        }
    }
}
```

If B is another 2-bit bus and B[0] is its MSB, the power relationships are as follows:

From pin	To pin	Label
B[0]	X[0]	B0_X0
B[0]	X[1]	B0_X1
B[0]	X[2]	B0_X2
B[0]	X[3]	B0_X3
B[1]	X[0]	B1_X0
B[1]	X[1]	B1_X1
B[1]	X[2]	B1_X2
B[1]	X[3]	B1_X3

internal_power Group

To define an `internal_power` group in a `pin` group, use these simple attributes, complex attributes, and groups:

Simple attributes:

- `equal_or_opposite_output`
- `falling_together_group`
- `related_pg_pin`
- `related_pin`
- `rising_together_group`
- `switching_interval`
- `switching_together_group`
- `when`

Complex attribute:

- `mode`

Groups:

- fall_power
- power
- rise_power

equal_or_opposite_output Simple Attribute

This attribute designates an optional output pin or pins whose capacitance can be used to access a three-dimensional table in the `internal_power` group.

Syntax

```
equal_or_opposite_output : "name | name_list" ;
```

name | name_list

Name of output pin or pins.

Note:

This pin (or these pins) have to be functionally equal to or the opposite of the pin named in the same `pin` group.

Example

```
equal_or_opposite_output : "Q" ;
```

Note:

The output capacitance of this pin (or pins) is used as the `equal_or_opposite_output_net_capacitance` value in the internal power lookup table.

falling_together_group Simple Attribute

Use the `falling_together_group` attribute to identify the list of two or more input or output pins that share logic and are falling together during the same time period. Set this time period with the `switching_interval` attribute. See “[switching_interval Simple Attribute](#)” on page 12-36 for details.

Together, the `falling_together_group` attribute and the `switching_interval` attribute settings determine the level of power consumption.

Define a `falling_together_group` attribute in the `internal_power` group in a `pin` group, as shown here.

```
cell (name_string) {
  pin (name_string) {
    internal_power () {
      falling_together_group : "list of pins" ;
```

```
    rising_together_group : "list of pins" ;
    switching_interval : float;
    rise_power () {
        ...
    }
    fall_power () {
        ...
    }
}
}
```

list of pins

The names of the input or output pins that share logic and are falling during the same time period.

related_pin Simple Attribute

This attribute associates the `internal_power` group with a specific input or output pin. If `related_pin` is an output pin, it must be functionally equal to or the opposite of the pin in that `pin` group.

If `related_pin` is an input pin or output pin, the pin's transition time is used as a variable in the internal power lookup table.

Syntax

```
related pin : "name | name list" ;
```

name | *name list*

Name of the input or output pin or pins

Example

```
related pin : "A B" ;
```

The pin or pins in the `related_pin` attribute denote the path dependency for the `internal_power` group. The Library Compiler tool accesses a particular `internal_power` group if the input pin or pins cause the corresponding output pin named in the `pin` group to toggle.

If you want to define a two-dimensional or three-dimensional table, specify all functionally related pins in a `related pin` attribute.

rising_together_group Simple Attribute

The `rising_together_group` attribute identifies the list of two or more input or output pins that share logic and are rising during the same time period. This time period is defined with the `switching_interval` attribute. See the following “[switching_interval Simple Attribute](#),” section for details.

Together, the `rising_together_group` attribute and `switching_interval` attribute settings determine the level of power consumption.

Define the `rising_together_group` attribute in the `internal_power` group, as shown here.

```
cell (name_string) {
    pin (name_string) {
        internal_power () {
            falling_together_group : "list of pins" ;
            rising_together_group : "list of pins" ;
            switching_interval : float;
            rise_power () {
                ...
            }
            fall_power () {
                ...
            }
        }
    }
}
```

list of pins

The names of the input or output pins that share logic and are rising during the same time period.

switching_interval Simple Attribute

The `switching_interval` attribute defines the time interval during which two or more pins that share logic are falling, rising, or switching (either falling or rising) during the same time period.

Set the `switching_interval` attribute together with the `falling_together_group`, `rising_together_group`, or `switching_together_group` attribute. Together with one of these attributes, the `switching_interval` attribute defines a level of power consumption.

For details about the attributes that are set together with the `switching_interval` attribute, see “[falling_together_group Simple Attribute](#)” on page 12-34; “[rising_together_group Simple Attribute](#)” on page 12-36; and the following “[switching_together_group Simple Attribute](#),” section.

Syntax

```
switching_interval : float ;
```

float

A floating-point number that represents the time interval during which two or more pins that share logic are switching together.

Example

```
pin (Z) {
    direction : output;
    internal_power () {
        switching_together_group : "A B";
        /*if pins A, B, and Z switch*/
        switching_interval : 5.0;
        /*switching within 5 time units */;
        power () {
            ...
        }
    }
}
```

switching_together_group Simple Attribute

The `switching_together_group` attribute identifies the list of two or more input or output pins that share logic, are either falling or rising during the same time period, and are not affecting power consumption.

Define the time period with the `switching_interval` attribute. See “[“switching_interval Simple Attribute” on page 12-36](#) for details.

Define the `switching_together_group` attribute in the `internal_power` group, as shown here.

```
cell (name_string) {
    pin (name_string) {
        internal_power () {
            switching_together_group : "list of pins" ;
            switching_interval : float;
            power () {
                ...
            }
        }
    }
}
```

list of pins

The names of the input or output pins that share logic, are either falling or rising during the same time period, and are not affecting power consumption.

when Simple Attribute

This attribute specifies a state-dependent condition that determines whether the internal power table is accessed.

You can use the `when` attribute to define one-, two-, or three-dimensional tables in the `internal_power` group.

Syntax

```
when : "Boolean expression" ;
```

Boolean expression

Name or names of input and output pins, buses, and bundles with corresponding Boolean operators.

[Table 12-1 on page 12-9](#) lists the Boolean operators valid in a `when` statement.

Example

```
when : "A B" ;
```

The Library Compiler tool checks that the `when` and `related_pin` attributes do not contain the same pin, as they do in the following example:

Example 12-3 The Same Pin Specified in Both the when and related_pin Attributes

```
pin (Z) {
    function : "A & B & C";
    internal_power () {
        related_pin : "B";
        when : "!A & B";
        ...
    }
}
```

If the `when` attribute and the `related_pin` attribute contain the same pin, the Library Compiler tool generates an error message similar to the following:

```
Error: Line 183, The 'A' when condition includes the 'A'
related pin. (LIBG-215)
```

mode Complex Attribute

You define the `mode` attribute within an `internal_power` group. A `mode` attribute pertains to an individual pin. The pin is active when `mode` is instantiated with a name and a value. You can specify multiple instances of the `mode` attribute but only one instance for each pin.

Syntax

```
mode (mode_definition_name, mode_value);
```

The Library Compiler tool issues an error message if the `mode_definition_name` and `mode_value` strings are not already defined by a `mode_definition` group in the cell.

Example

```
cell (my_cell) {
    mode_definition (mode_definition_name) {
        mode_value(namestring) {
            when : "Boolean expression";
            sdf_cond : "Boolean expression";
        } ...
        ...
        pin (pin_name) {
            direction : output ;
            function : "Boolean expression";
            internal_power () {
                related_pin : "pin_name";
                /*when : "Boolean expression";*/
                mode (mode_definition_name, mode_value);
                ...
            }/* end internal_power() */
        }/* end pin */
    }/* end cell */
```

[Example 12-4](#) shows a mode instance description.

Example 12-4 A mode Instance Description

```
library (my_library) {
    technology ( cmos ) ;
    delay_model : table_lookup;
    ...
    cell(inv0d0) {
        area : 0.75;
        mode_definition(rw) {
            mode_value(read) {
                when : "A";
                sdf_cond : "A == 1";
            }
            mode_value(write) {
                when : "!A";
                sdf_cond : "A == 0";
            }
        }
        pin(A) {
            direction : input;
            max_transition : 2100.0;
            capacitance : 0.002000;
            fanout_load : 1;
            ...
        }
        pin(I) {
            direction : input;
            max_transition : 2100.0;
            capacitance : 0.002000;
            fanout_load : 1;
```

```

    ...
}

pin(Z) {
    direction : output;
    max_capacitance : 0.175000;
    max_fanout : 58;
    max_transition : 1400.0;
    function : "(I)'";
    internal_power () {
        related_pin : "I";
        mode(rw, read);
        /* when : "A"; */
        ...
    }
    internal_power () {
        related_pin : "I";
        mode(rw, write);
        /* when : "!A"; */
        ...
    }
    timing() {
        related_pin : "I";
        timing_sense : negative_unate;
        ...
        } /* timing I -> Z */
    } /* I */
    ...
} /* cell(inv0d0) */
} /* library */

```

fall_power Group

Use a `fall_power` group to define a fall transition for a pin. If you specify a `fall_power` group, you must also specify a `rise_power` group. However, if the related pin of the `internal_power` group is a preset or clear pin, you can specify either only the `fall_power` or only the `rise_power` group.

You define a `fall_power` group in an `internal_power` group in a cell-level pin group, as shown here:

```

cell (name_string) {
    pin (name_string) {
        internal_power () {
            fall_power (template name) {
                ... fall power description ...
            }
        }
    }
}

```

Complex Attributes

```

index_1 ("float, ..., float") ; /* lookup table */
index_2 ("float, ..., float") ; /* lookup table */
index_3 ("float, ..., float") ; /* lookup table */

```

```
values ("float, ..., float") ; /* lookup table */
```

index_1, index_2, index_3 Attributes

These attributes identify internal cell consumption per fall transition. Define these attributes in the `internal_power` group or in the library-level `power_lut_template` group.

values Attribute

This attribute defines internal cell consumption per fall transition.

Example

```
values ("2.2, 3.7, 4.3", "1.7, 2.1, 3.5", "1.0, 1.5, 2.8");
```

You must ensure that the internal power units exactly match the derived units of switching power (CV^2). This is necessary because the Design Compiler tool adds the `internal_power` value and switching power to get the dynamic power consumption without any consideration of units.

The Design Compiler tool converts the `values` attribute information to power consumption by multiplying the unit by the factor transition or `per_unit_time`, as shown here:

- `nindex_1` floating-point numbers if the lookup table is one-dimensional
- `nindex_1 x nindex_2` floating-point numbers if the lookup table is two-dimensional
- `nindex_1 x nindex_2 x nindex_3` floating-point numbers if the lookup table is three-dimensional

The `nindex_1`, `nindex_2`, and `nindex_3` numbers are the size of `index_1`, `index_2`, and `index_3` in this group or in the `power_lut_template` group it inherits. Quotation marks enclose a group. Each group represents a row in the table.

power Group

The `power` group is defined within an `internal_power` group in a `pin` group at the cell level, as shown here:

```
library (name) {
  cell (name) {
    pin (name) {
      internal_power () {
        power (template name) {
          ... power template description ...
        }
      }
    }
  }
}
```

Complex Attributes

```
index_1 ("float, ..., float") ; /* lookup table */
index_2 ("float, ..., float") ; /* lookup table */
index_3 ("float, ..., float") ; /* lookup table */
values ("float, ..., float") ; /* lookup table */
```

index_1, index_2, index_3 Attributes

These attributes identify internal cell consumption per transition. Define these attributes in the `internal_power` group or in the library-level `power_lut_template` group.

values Attribute

This attribute defines internal cell power consumption per rise or fall transition.

This power information is accessed when the pin has a rise transition or a fall transition. The values in the table specify the average power per transition.

Note:

The internal power value is derived from the capacitance unit exponent (with mantissa discarded) and the voltage unit.

rise_power Group

A `rise_power` group is defined in an `internal_power` group at the cell level, as shown here:

```
cell (name) {
    pin (name) {
        internal_power () {
            rise_power (template name) {
                ... rise power description ...
            }
        }
    }
}
```

Rise power is accessed when the pin has a rise transition. If you have a `rise_power` group, you must have a `fall_power` group. However, if the related pin of the `internal_power` group is a preset or clear pin, you can specify either only the `fall_power` or only the `rise_power` group.

Complex Attributes

```
index_1 ("float, ..., float") ; /* lookup table */
index_2 ("float, ..., float") ; /* lookup table */
index_3 ("float, ..., float") ; /* lookup table */
values ("float, ..., float") ; /* lookup table */
```

index_1, index_2, index_3 Attributes

These attributes identify internal cell consumption per rise transition. Define these attributes in the `internal_power` group or in the library-level `power_lut_template` group.

values Attribute

This attribute defines internal cell power consumption per rise transition.

Note:

The Design Compiler tool converts the `values` attribute information to power consumption, by multiplying the unit by the factor `transition` or `per_unit_time`. For more information, see “[values Attribute](#)” on page 12-41.

Syntax for One-Dimensional, Two-Dimensional, and Three-Dimensional Tables

You can define a one-, two-, or three-dimensional table in the `internal_power` group in either of the following two ways:

- Using the `power` group. For two- and three-dimensional tables, define the `power` group and the `related_pin` attribute.
- Using a combination of the `related_pin` attribute, the `fall_power` group, and the `rise_power` group.

The syntax for a one-dimensional table using the `power` group is shown here:

```
internal_power() {
    power (template name) {
        values("float, ..., float");
    }
}
```

The syntax for a one-dimensional table using the `fall_power` and `rise_power` groups is shown here:

```
internal_power() {
    fall_power (template name) {
        values("float, ..., float");
    }

    rise_power (template name) {
        values("float, ..., float");
    }
}
```

The syntax for a two-dimensional table using the `power` and `related_pin` groups is shown here:

```
internal_power() {
    related_pin : "name | name_list" ;
```

```

power (template name) {
    values("float, ..., float") ;
}
}

```

The syntax for a two-dimensional table using the `related_pin`, `fall_power`, and `rise_power` groups is shown here:

```

internal_power() {
    related_pin : "name | name_list" ;
    fall_power (template name) {
        values("float, ..., float");
    }
    rise_power (template name) {
        values("float, ..., float");
    }
}

```

The syntax for a three-dimensional table using the `power` and `related_pin` groups is shown here:

```

internal_power() {
    related_pin : "name | name_list" ;
    power (template name) {
        values("float, ..., float") ;
    }
    equal_or_opposite_output : "name | name_list" ;
}

```

The syntax for a three-dimensional table using the `related_pin`, `fall_power`, and `rise_power` groups is shown here:

```

internal_power() {
    related_pin : "name | name_list" ;
    fall_power (template name) {
        values ("float, ..., float") ;
    }
    rise_power (template name) {
        values ("float, ..., float") ;
    }
    equal_or_opposite_output : "name | name_list" ;
}

```

[Example 12-5](#) shows cells that contain internal power information in the `pin` group.

Example 12-5 A Library With Internal Power

```

library(internal_power_example) {
    ...
    power_lut_template(output_by_cap_and_trans) {
        variable_1 : total_output_net_capacitance ;
        variable_2 : input_transition_time ;
        index_1 ("0.0, 5.0, 20.0");
        index_2 ("0.0, 1.0, 20.0");
    }
}

```

```

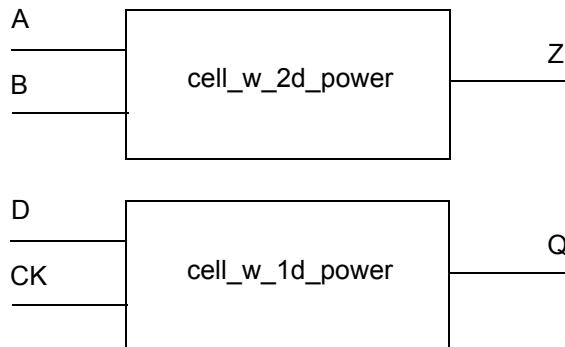
}
...
cell(AN2) {
    pin(Z) {
        direction : output ;
        internal_power {
            power(output_by_cap_and_trans) {
                values ("2.2, 3.7, 4.3", "1.7, 2.1, 3.5", "1.0, 1.5, 2.8");
            }
            related_pin : "A B" ;
        }
    }
    pin(A) {
        direction : input ;
        ...
    }
    pin(B) {
        direction : input ;
        ...
    }
}
}

```

Internal Power Examples

The examples in this section show how to describe the internal power of the 2-input sequential gate in [Figure 12-6](#), using one-, two-, and three-dimensional lookup tables.

Figure 12-6 Library Cells With Internal Power Information



One-Dimensional Power Lookup Table

[Example 12-6](#) is the library description of the cell shown in [Figure 12-6](#), a cell with a one-dimensional internal power table defined in the `pin` groups.

Example 12-6 One-Dimensional Internal Power Table

```

library(internal_power_example) {
    ...

```

```

power_lut_template(output_by_cap) {
    variable_1 : total_output_net_capacitance ;
    index_1("0.0, 5.0, 20.0") ;
}
...
power_lut_template(input_by_trans) {
    variable_1 : input_transition_time ;
    index_1 ("0.0, 1.0, 2.0") ;
}
...
cell(FLOP1) {
    pin(CP) {
        direction : input ;
        internal_power()
            power (input_by_trans) {
                values("1.5, 2.6, 4.7") ;
            }
    }
}
...
pin(Q) {
    direction : input ;
    internal_power() {
        power(output_by_cap) {
            values("9.0, 5.0, 2.0") ;
        }
    }
}
}
}

```

In Example 12-6, the input transition time at pin CP is 1.0. The output load at pin Q is 5.0. The toggle rate at pin CP is two transitions per 100 ns. The toggle rate at pin Q is one transition per 100 ns. Using this information to index into the table, you get the following results:

$$\begin{aligned} E_{CP} &= 2.6 \\ E_Q &= 5.0 \end{aligned}$$

If the `time_unit` attribute value in the technology library is 1 ns, the total internal power consumed by this gate is

$$\begin{aligned} P_{int} &= (E_{CP} \times AF_{CP}) + (E_Q \times AF_Q) \\ &= (2.6 \times 2 \times 10^{-2}) + (5.0 \times 1 \times 10^{-2}) \\ &= 0.102 \end{aligned}$$

The activity factors AF_{CP} and AF_Q are adjusted to the `time_unit` specified in the technology library. The unit of P_{int} depends on the unit of the `voltage_unit` and capacitive load unit attributes in the technology library.

If the following attributes and values are specified in the technology library, the unit of the internal power group is 10^{-15} joule/transition.

```
voltage_unit : "1V";
capacitive_load_unit(1.0, "ff");
```

Because the unit of the activity factor is transition/ns, the total internal power dissipation of this gate is

$$0.102 \times 10^{-15} \text{ joule/transition} \times \text{transition}/10^{-9}\text{second} = 0.102 \text{ uW.}$$

Note:

For a detailed explanation of how to calculate internal power, see the *Power Compiler User Guide*.

Two-Dimensional Power Lookup Table

[Example 12-7](#) is the library description of the cell in [Figure 12-6](#), a cell with a two-dimensional internal power table defined in the pin groups.

Example 12-7 Two-Dimensional Internal Power Table

```
library(internal_power_example) {
    ...
    power_lut_template(output_by_cap_and_trans) {
        variable_1 : total_output_net_capacitance ;
        variable_2 : input_transition_time ;
        index_1 ("0.0, 5.0, 20.0");
        index_2 ("0.0, 1.0, 20.0");
    }
    ...
    cell(AN2) {
        pin(Z) {
            direction : output ;
            internal_power {
                power(output_by_cap_and_trans) {
                    values ("2.2, 3.7, 4.3", "1.7, 2.1, 3.5", "1.0, 1.5, 2.8");
                }
                related_pin : "A B" ;
            }
        }
        pin(A) {
            direction : input ;
            ...
        }
        pin(B) {
            direction : input ;
            ...
        }
    }
}
```

In this example, the input transition time at pin A is 1.25. The input transition time at pin B is 2.5. The output load at pin Z is 5.0. The toggle rate at pin A is 2 transitions per 100 ns. The toggle rate at pin B is three transitions per 100 ns. The toggle rate at pin Z is one transition per 100 ns.

With this information, the weighted input transition time is calculated as follows. (In this case, you use the weighted input transition time because there are two inputs.)

$$(1.25 \times 2 \times 10^{-2} + 2.5 \times 3 \times 10^{-2}) / (2 \times 10^{-2} + 3 \times 10^{-2}) = 2.0$$

Because the output load at pin Z is 5.0, the following power table values are used:

$$\begin{aligned} 2.1 &= 1.0A + B \\ 3.5 &= 20.0A + B \end{aligned}$$

Consequently, the values for A and B become

$$\begin{aligned} A &= 0.0739 \\ B &= 2.0263 \end{aligned}$$

and the resulting values for Ez and Pint are

$$\begin{aligned} Ez &= 2.0 \times 0.0739 + 2.0263 \\ &= 2.1741 \\ &= 2.2 \end{aligned}$$

$$\begin{aligned} P_{int} &= Ez \times AF_Z \\ &= 2.2 \times 1 \times 10^{-E2} \\ &= 0.022 \end{aligned}$$

The activity factor AF_Z is adjusted to the `time_unit` specified in the technology library. The unit of `internal_power` can be applied here.

Three-Dimensional Power Lookup Table

[Example 12-8](#) is the library description for the cell in [Figure 12-6](#), a cell with a three-dimensional internal power table.

Example 12-8 Three-Dimensional Internal Power Table

```
library(internal_power_example) {
  ...
  power_lut_template(output_by_cap1_cap2_and_trans) {
    variable_1 : total_output1_net_capacitance ;
    variable_2 : equal_or_opposite_output_net_capacitance ;
    variable_3 : input_transition_time ;
    index_1 ("0.0, 5.0, 20.0") ;
    index_2 ("0.0, 5.0, 20.0") ;
    index_3 ("0.0, 1.0, 2.0") ;
  }
  ...
  cell(FLOP1) {
    pin(CP) {
      direction : input ;
      ...
    }
    pin(D) {
      direction : input ;
      ...
    }
  }
}
```

```

pin(S) {
    direction : input ;
    ...
}
pin(R) {
    direction : input ;
    ...
}
pin(Q) {
    direction : output ;
    internal_power() {
        power(output_by_cap1_cap2_and_trans) {
            values("2.2, 3.7, 4.3", "1.7, 2.1, 3.5", "1.0, 1.5,\n
                    .8", "2.1, 3.6, 4.2", "1.6, 2.0, 3.4", "0.9, 1.5, .7" \
                    "2.0, 3.5, 4.1", "1.5, 1.9, 3.3", "0.8, 1.4,2.6");
        }
        equal_or_opposite_output : "QN" ;
        related_pin : "CP" ;
    }
    ...
}
...
}
}

```

In this example, the input transition time at pin CP is 1.0. The output load at pin QN is 5.0. The output load at pin Q is 5.0. The toggle rate at pin Q is one transition per 100 ns.

Using the transition time and the loading at pin Q and QN to index into the table, you get the following results:

$$EQ = 2.0$$

If the `time_unit` in the technology library is 1 ns, the total internal power consumed by this gate is

$$\begin{aligned} P_{int} &= EQ \times AF_Q \\ &= 2.0 \times 1 \times 10^{-2} \\ &= 0.020 \end{aligned}$$

The activity factor AF_Q is adjusted to the `time_unit` specified in the technology library. The unit of `internal_power` can be applied here.

Multiple Power Supply Library Requirements

The Library Compiler tool confirms that a library with multiple power supplies meets the following requirements:

- In any cell with two or more power supplies, all the pins, buses, and bundles have an `related_power_pin` and `related_power_pin` attribute pair for input and output direction signals.

- In any cell with two or more power supplies, all the I/O pins, buses, and bundles have an `related_power_pin` and an `related_power_pin` attribute.
- An output pin within a cell that has multiple power supplies requires `internal_power` tables with the `related_pg_pin` attribute for all the different PG pins of the cell.
- An input pin within a cell that has multiple power supplies requires an `internal_power` table with the `related_pg_pin` value matching the `related_power_pin` value.

Modeling Libraries With Integrated Clock-Gating Cells

Power optimization achieved 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 power.

You can perform automatic clock gating at the register transfer level, using the HDL Compiler and Power Compiler tools from Synopsys.

What Clock Gating Does

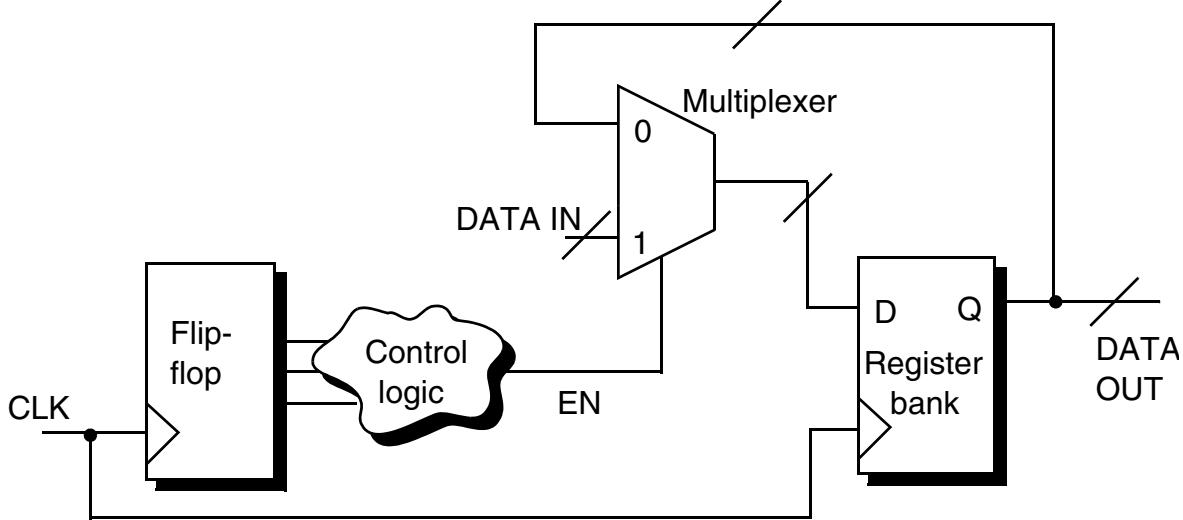
Clock gating provides a power-efficient implementation of register banks that are disabled during some clock cycles.

A register bank is a group 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.

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

[Figure 12-7](#) shows a simple implementation of a register bank using a multiplexer and a feedback loop.

Figure 12-7 Synchronous Load-Enable Register Using a 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, allowing new values to load at the D input.

Such feedback loops can use some power unnecessarily. 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.

By controlling the clock signal for the register, you can eliminate the need for reloading the same value in the register through multiple clock cycles. Clock gating inserts a 2-input gate into the register bank's clock network, creating the control to eliminate unnecessary register activity.

Clock gating reduces the clock network's power dissipation and often relaxes the datapath timing. If your design has large multibit 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 integrated clock-gating cell functionality, you have the option of doing the following:

- Use latch-free or latch-based clock gating.
- Insert logic to increase testability.

For details, see “[Using an Integrated Clock-Gating Cell](#)” and “[Setting Pin Attributes for an Integrated Cell](#)” on page 12-53.

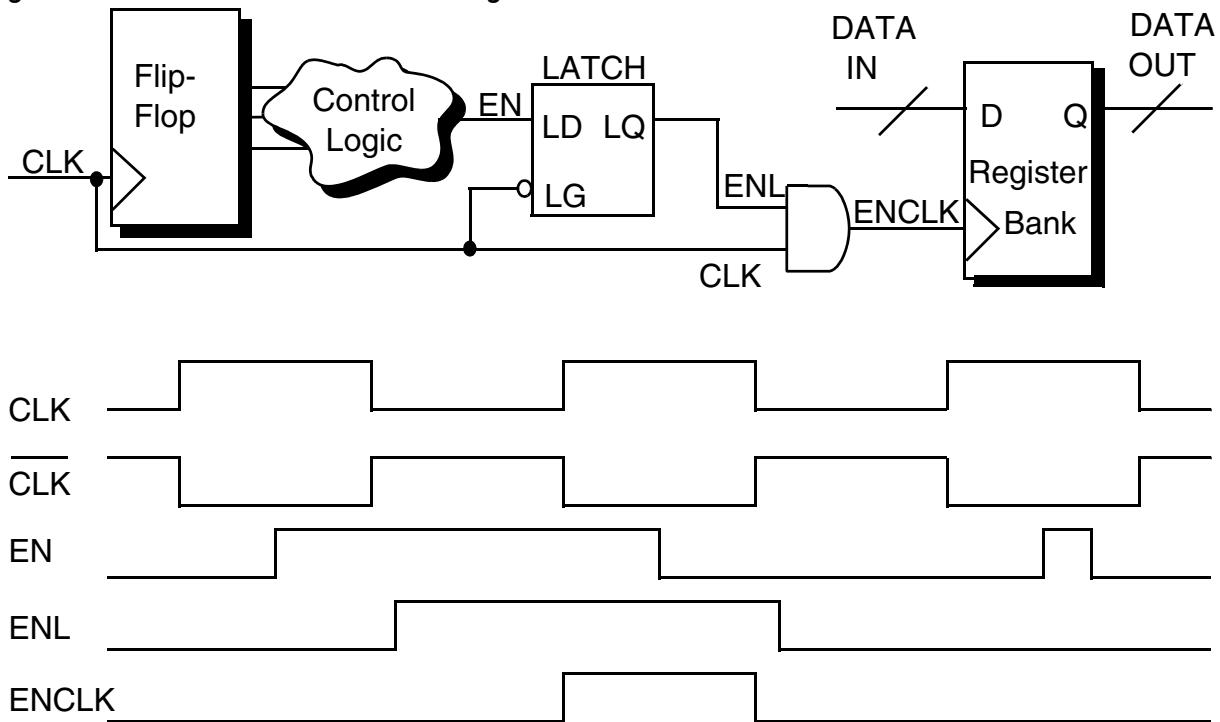
Looking at a Gated Clock

Clock gating saves power by eliminating the unnecessary activity associated with reloading register banks. Clock gating eliminates the feedback net and multiplexer shown in [Figure 12-7](#), by inserting a 2-input gate in the clock net of the register. The 2-input clock gate selectively prevents clock edges, thus preventing the gated clock signal from clocking the gated register.

Clock gating can also insert inverters or buffers to satisfy timing or clock waveform and duty requirements.

[Figure 12-8](#) shows the 2-input clock gate as an AND gate; however, depending on the type of register and the gating style, gating can use NAND, OR, and NOR gates instead. At the bottom of Figure 12-8, the waveforms of the signals are shown with respect to the clock signal, CLK.

Figure 12-8 Latch-Based Clock Gating



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 12-7](#). The register 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 positive-edge clocks, the clock gate forces the clock-gated signal to maintain logic state 0 after the falling edge of the clock.

Using an Integrated Clock-Gating Cell

Consider using an integrated clock-gating cell if you are experiencing timing problems caused by the introduction of random logic on the clock line.

Create an integrated clock-gating cell that integrates the various combinational and sequential elements of the clock-gating circuitry into a single cell, which is compiled to gates and located in the technology library.

The Library Compiler tool recognizes a clock-gating cell by accessing the state tables and state functions of the library cell pins.

Setting Pin Attributes for an Integrated Cell

The clock-gating software requires the pins of your integrated cells to be set with the attributes listed in [Table 12-3](#). Setting some of the pin attributes, such as those for test and observability, is optional.

Table 12-3 Pin Attributes for Integrated Clock-Gating Cells

Integrated cell pin name	Data direction	Required attribute
clock	in	clock_gate_clock_pin
enable	in	clock_gate_enable_pin
test_mode or scan_enable	in	clock_gate_test_pin
observability	out	clock_gate_obs_pin
enable_clock	out	clock_gate_out_pin

For more details about the `clock_gating_integrated_cell` attribute and the corresponding pin attributes, see the *Power Compiler User Guide*.

clock_gate_clock_pin Attribute

The `clock_gate_clock_pin` attribute identifies an input pin connected to a clock signal. Design Compiler uses the `clock_gate_clock_pin` attribute when it compiles a design containing gated clocks that were introduced by Power Compiler.

Syntax

```
clock_gate_clock_pin : true | false ;
```

true | false

A true value labels the pin as a clock pin. A false value labels the pin as *not* a clock pin.

Example

```
clock_gate_clock_pin : true;
```

clock_gate_enable_pin Simple Attribute

Design Compiler uses the `clock_gate_enable_pin` attribute when it compiles a design containing gated clocks that were introduced by Power Compiler.

The `clock_gate_enable_pin` attribute identifies an input pin connected to an enable signal for nonintegrated clock-gating cells and integrated clock-gating cells.

Syntax

```
clock_gate_enable_pin : true | false ;
```

true | false

A true value labels the input pin of a clock-gating cell connected to an enable signal as the enable pin. A false value does not label the input pin as an enable pin.

Example

```
clock_gate_enable_pin : true;
```

For clock-gating cells, you can set this attribute to `true` on only one input port of a 2-input AND, NAND, OR, or NOR gate. If you do so, the other input port is the clock.

For more information about identifying pins on integrated clock-gating cells, see “[Using an Integrated Clock-Gating Cell](#)” on page 12-53.

For additional information about integrated clock gating, see the *Power Compiler User Guide*.

clock_gate_test_pin Attribute

The `clock_gate_test_pin` attribute identifies an input pin connected to a `test_mode` or `scan_enable` signal. This attribute is used by Design Compiler when it compiles a design containing gated clocks that were introduced by Power Compiler.

Syntax

```
clock_gate_test_pin : true | false ;
```

true | false

A true value labels the pin as a test (`test_mode` or `scan_enable`) pin. A false value labels the pin as *not* a test pin.

Example

```
clock_gate_test_pin : true;
```

clock_gate_obs_pin Attribute

The `clock_gate_obs_pin` attribute identifies an output pin connected to an observability signal. Design Compiler uses the `clock_gate_obs_pin` attribute when it compiles a design containing gated clocks that were introduced by Power Compiler.

Syntax

```
clock_gate_obs_pin : true | false ;
```

true | false

A true value labels the pin as an observability pin. A false value labels the pin as *not* an observability pin.

Example

```
clock_gate_obs_pin : true;
```

clock_gate_out_pin Attribute

The `clock_gate_out_pin` attribute identifies an output port connected to an `enable_clock` signal. The Design Compiler tool uses the `clock_gate_out_pin` attribute when it compiles a design containing gated clocks that were introduced by the Power Compiler tool.

Syntax

```
clock_gate_out_pin : true | false ;
```

true | false

A true value labels the pin as a clock-gated output (`enable_clock`) pin. A false value labels the pin as *not* a clock-gated output pin.

Example

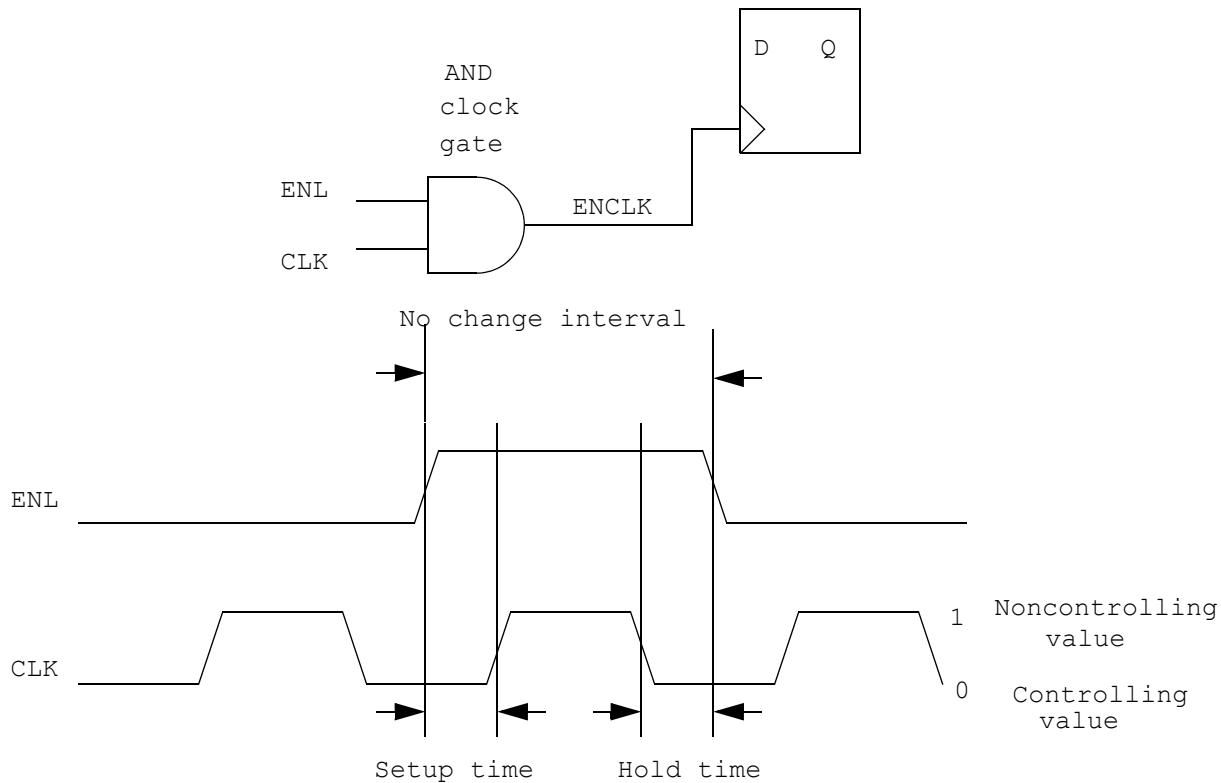
```
clock_gate_out_pin : true;
```

Clock-Gating Timing Considerations

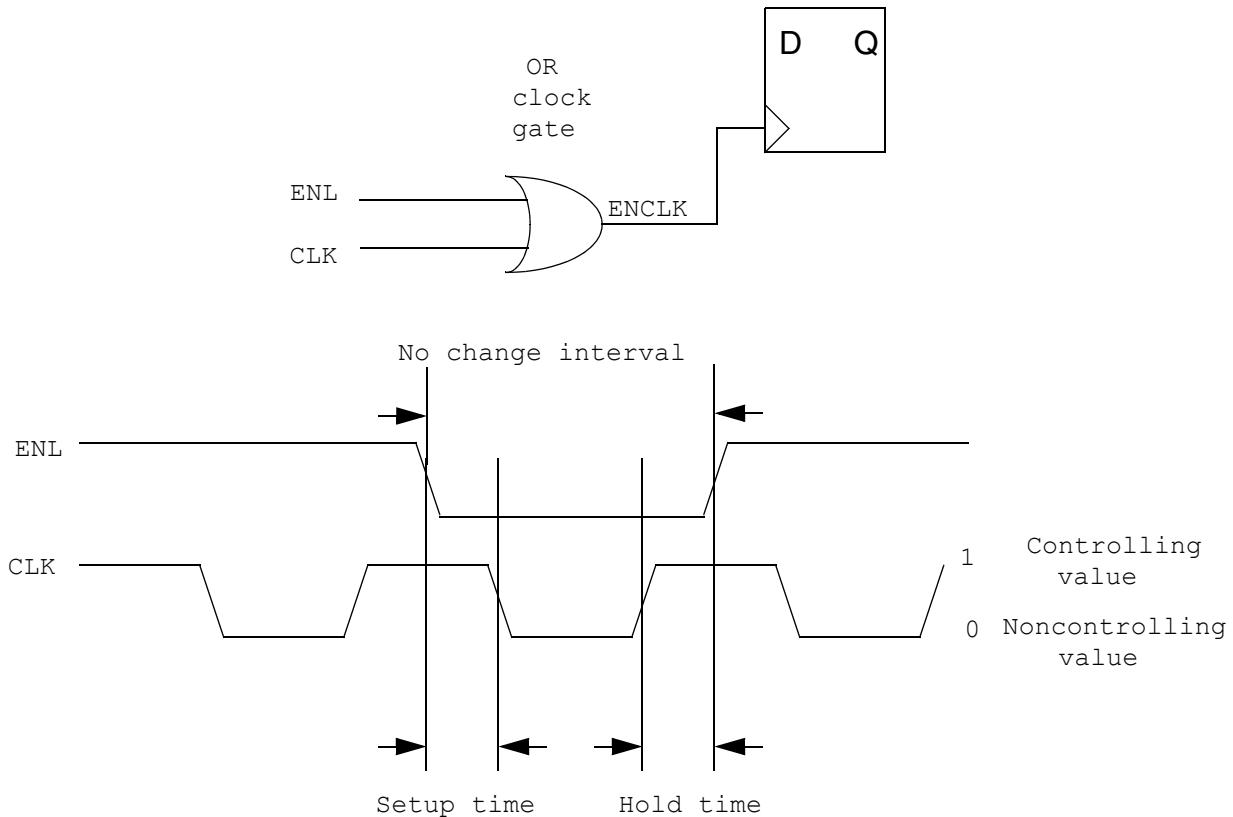
The clock gate must not alter the waveform of the clock—other than turning the clock signal on and off.

[Figure 12-9](#) and [Figure 12-10](#) show the relationship of setup and hold times to a clock waveform. [Figure 12-9](#) shows the relationship when an AND gate is the clock-gating element. [Figure 12-10](#) shows the relationship when an OR gate is the clock-gating element.

Figure 12-9 Setup and Hold Times for an AND Clock Gate



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

Figure 12-10 Setup and Hold Times for an OR Clock Gate

Timing Considerations for Integrated Cells

Clock gating requires certain timing arcs on your integrated cell.

For latch-based clock gating,

- Define setup and hold arcs on the enable pins with respect to the same controlling edge of the clock.
- Define combinational arcs from the clock and enable inputs to the output.

For latch-free clock gating,

- Define no-change arcs on the enable pins. 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.

Table 12-4 specifies the setup and hold arcs required on the integrated cells. Set the setup and the hold arcs on the enable pin as specified by the `clock_gate_enable_pin` attribute with respect to the value entered for the `clock_gating_integrated_cell` attribute.

For the latch- and flip-flop-based styles, the setup and hold arcs are the conventional type and are set with respect to the same clock edge. However, for the latch-free style, the setup and hold arcs are set with respect to different clock edges and therefore must be specified as no-change arcs. Note that all arcs for integrated cells must be combinational arcs.

Table 12-4 Values of the `clock_gating_integrated_cell` Attribute

<code>clock_gating_integrated_cell</code> attribute value	Setup arc	Hold arc
<code>latch_posedge</code>	rising	rising
<code>latch_negedge</code>	falling	falling
<code>none_posedge</code>	falling	rising
<code>none_negedge</code>	rising	falling
<code>ff_posedge</code>	falling	falling
<code>ff_negedge</code>	rising	rising

In checking the timing arcs on integrated clock-gating cells, the Library Compiler tool does the following:

- If a combinational integrated clock-gating cell or simple clock gating cell has sequential setup and hold arcs, the Library Compiler tool issues no warning.
- If a sequential integrated clock-gating cell has combinational arcs from the inputs to the outputs, the Library Compiler tool issues no warning.
- If there is no setup or hold on the enable pin from the clock pin, the Library Compiler tool issues an error.
- If there is no combinational arc from the clock to the output, the Library Compiler tool issues an error message. This combinational arc is needed to propagate the clock properties from inputs to outputs.
- If there is a sequential arc from the clock or enable signal to the output pin, the Library Compiler tool issues an error message. This arc prevents propagation of clock properties to the output.

Integrated Clock-Gating Cell Example

Example 12-9 shows what you might enter in setting up a cell for integrated clock gating.

This example uses the `latch_posedge_precontrol_obs` value option for the `clock_gating_integrated_cell` attribute.

Example 12-9 Integrated Clock-Gating Cell

```
cell(CGLPC0) {
    area : 1;
    clock_gating_integrated_cell : "latch_posedge_precontrol_obs";
    dont_use : true;
    statetable(" CLK EN SE", "IQ") {
        table : " L  L  L : - : L , \
                  L  L  H : - : H , \
                  L  H  L : - : H , \
                  L  H  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;
            cell_rise(scalar) {
                values( " 0.4 ");
            }
            cell_fall(scalar) {
                values( " 0.4 ");
            }
            related_pin : "CLK";
        }
        timing() {
            timing_type : hold_rising;
            cell_rise(scalar) {
                values( " 0.4 ");
            }
            cell_fall(scalar) {
                values( " 0.4 ");
            }
            related_pin : "CLK";
        }
    }
    pin(SE) {
        direction : input;
        capacitance : 0.017997;
        clock_gate_test_pin : true;
    }
    pin(CLK) {
        direction : input;
        capacitance : 0.031419;
        clock_gate_clock_pin : true;
        min_pulse_width_low : 0.319;
    }
    pin(GCLK) {
        direction : output;
        state_function : "CLK * IQ";
        max_capacitance : 0.500;
        clock_gate_out_pin : true;
    }
}
```

```

timing() {
    timing_sense : positive_unate;
    cell_rise(scalar) {
        values( " 0.4 ");
    }
    rise_transition(scalar) {
        values( " 0.1 ");
    }
    cell_fall(scalar) {
        values( " 0.4 ");
    }
    fall_transition(scalar) {
        values( " 0.1 ");
    }
    related_pin : "EN CLK";
}
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("0.141, 0.148, 0.256", \
        "0.162, 0.145, 0.234", \
        "0.192, 0.200, 0.284", \
        "0.199, 0.219, 0.297");
    }
    related_pin : "CLK EN" ;
}
pin(OBS) {
    direction : output;
    state_function : "EN";
    max_capacitance : 0.500;
    clock_gate_obs_pin : true;
}
}

```

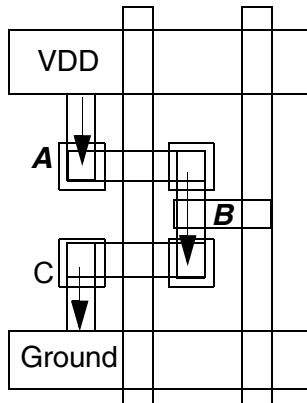
Modeling Electromigration

When high-density current passes through a thin metal wire, the high-energy electrons exert forces upon the atoms that causes electromigration of the atoms. Electromigration can drastically reduce the lifetime of a chip by increasing the resistivity of the metal wire or creating a short circuit between adjacent lines.

Overview

[Figure 12-11](#) shows a CMOS inverter layout. The connections between the drains of the NMOS and PMOS transistors (section B), and between the sources and VDD and ground (sections A and C) are typical examples of wires exposed to unidirectional current with a pulse rate equal to the output toggle rate.

Figure 12-11 Electromigration in an Inverter



Electromigration failure strongly depends on the toggle rate.

The reliability of a metal line is most commonly described by the mean time to failure (MTTF) based on a set of experiments for a set of wires. The general expression for MTTF is

$$\text{MTTF} \equiv AJ^{-n} \exp(E_a/kT)$$

A

Material constant based on microstructure and geometric properties.

J

Average current density.

n

Constant whose value ranges between 1 and 3, representing the current density dependence.

E_a

Activation energy.

k

Boltzmann's constant.

T

Temperature in Kelvin.

To meet electromigration requirements of a cell, you can constrain each output rate to a maximal output toggle rate.

For the purposes of verification, each cell output is characterized with a multidimensional table in which the dimensions are output load (C_l), average input transition times, rising and falling (T_{rf}), temperature in Kelvin (T), and the cell lifetime (L). Some of these parameters can be fixed at specific values, decreasing the number of dimensions in the characterization table. The most common table is the two-dimension table, C_l and T_{rf} . The stored values are the maximal toggle rates.

Scaling of the electromigration constraints is performed for different operating temperatures. For enabling temperature scaling, a new factor, `em_temp_exp_degradation`, must be defined in the library. Consider the exponential factor in the previous section, $\exp(nE_a/kT)$; `em_temp_exp_degradation` represents its value for the nominal temperature. When the circuit operates in a nonnominal temperature, the electromigration constraints in the library is scaled with the following coefficient:

$$\begin{aligned} \text{temp_scale} &= \exp(nE_a k T_{\text{operating}}) / \exp(nE_a k T_{\text{nominal}}) = \\ &= \exp(nE_a k T_{\text{operating}}) / \text{em_temp_exp_degradation} = \\ &= \text{em_temp_exp_degradation}^{T_{\text{nominal}}/T_{\text{operating}} - 1} \end{aligned}$$

Controlling Electromigration

You can control electromigration by establishing an upper boundary for the output toggle rate. You achieve this by annotating cells with electromigration characterization tables representing the net's maximal toggle rates.

Specifically, do the following to control electromigration:

Use the `em_lut_template` group to name an index template to be used by the `em_max_toggle_rate` group, which you use to set the net's maximum toggle rates. The `em_lut_template` group has `variable_1`, `variable_2`, `index_1`, and `index_2` attributes.

- Use the `variable_1` and the `variable_2` attributes to specify the variables. Both `variable_1` and `variable_2` can take the `input_transition_time` or the `total_output_net_capacitance` value.
- Use the `index_1` attribute to specify the points along the `variable_1` table axis, and the `index_2` attribute to specify the points along the `variable_2` table axis.

The pin-level `electromigration` group contains the `em_max_toggle_rate` group, which you use to specify the maximum toggle rate and the `related_pin` and `related_bus_pins` attributes.

- Use the `related_pin` attribute to identify the input pin with which the `electromigration` group is associated.
- Use the `related_bus_pins` attribute to identify the `bus` group of the pin or pins with which the `electromigration` group is associated.
- To select the template you want the `em_max_toggle_rate` group to use, assign the name of the library-level `em_lut_template` group to the pin-level `em_max_toggle_rate` group.

The `em_max_toggle_rate` group contains the `values` complex attribute; the `related_pin` and `related_bus_pins` simple attributes; and, optionally, the `index_1` and `index_2` complex attributes.

- Use the `values` complex attribute to specify the nets' maximum toggle rates for every `index_1` breakpoint along the `variable_1` axis if the table is one-dimensional.
- If the table is two-dimensional, use `values` to specify the nets' maximum toggle rates for all points where the breakpoints of `index_1` intersect with the breakpoints of `index_2`. The value for these points is equal to $n_{index_1} \times n_{index_2}$, where `index_1` and `index_2` are the size to which the `em_lut_template` group's `index_1` and `index_2` attributes are set.
- You can also use the `em_max_toggle_rate` group's optional `index_1` and `index_2` attributes to overwrite the `em_lut_template` group's `index_1` and `index_2` values.
- State dependency in both lookup tables and polynomials.

Use the optional `em_temp_degradation_factor` at the library level or the cell level to specify the electromigration temperature exponential degradation factor. If this factor is not defined, the nominal temperature electromigration tables are used for all operating temperatures.

Use the `report_lib -em` command to get a report on electromigration for a specified library.

em_lut_template Group

Use the `em_lut_template` group at the library level to specify the name of the index template to be used by the `em_max_toggle_rate` group, which you use to set the net's maximum toggle rates to control electromigration.

Syntax

```
library (name) {
    em_lut_template(name_string) {
        variable_1: input_transition_time | total_output_net_capacitance ;
        variable_2: input_transition_time | total_output_net_capacitance ;
        index_1("float, ..., float");
        index_2("float, ..., float");
    }
}
```

variable_1 and variable_2 Simple Attributes

Use `variable_1` to assign values to the first dimension and `variable_2` to assign values to the second dimension for the templates on electromigration tables. The values can be either `input_transition_time` or `total_output_net_capacitance`.

Syntax

```
variable_1: input_transition_time |
total_output_net_capacitance ;
variable_2: input_transition_time |
total_output_net_capacitance ;
```

The value you assign to `variable_1` is determined by how the `index_1` complex attribute is measured, and the value you assign to `variable_2` is determined by how the `index_2` complex attribute is measured.

Assign `input_transition_time` for `variable_1` if the complex attribute `index_1` is measured with the input net transition time of the pin specified in the `related_pin` attribute or the pin associated with the electromigration group. Assign `total_output_net_capacitance` to `variable_1` if the complex attribute `index_1` is measured with the loading of the output net capacitance of the pin associated with the `em_max_toggle_rate` group.

Assign `input_transition_time` for `variable_2` if the complex attribute `index_2` is measured with the input net transition time of the pin specified in the `related_pin` attribute, in the `related_bus_pins` attribute, or in the pin associated with the electromigration group.

Assign `total_output_net_capacitance` to `variable_2` if the complex attribute `index_2` is measured with the loading of the output net capacitance of the pin associated with the electromigration group.

Example

```
variable_1 : total_output_net_capacitance ;
variable_2 : input_transition_time ;
```

index_1 and index_2 Complex Attributes

You can use the `index_1` optional attribute to specify the breakpoints of the first dimension of an electromigration table used to characterize cells for electromigration within the library. You can use the `index_2` optional attribute to specify the breakpoints of the second dimension of an electromigration table used to characterize cells for electromigration within the library.

Syntax

```
index_1 : ("float, ..., float") ;
index_2 : ("float, ..., float") ;
```

float

Floating-point numbers that identify the maximum toggle rate frequency from 1 to 0 and from 0 to 1.

You can overwrite the values entered for the `em_lut_template` group's `index_1`, by entering a value for the `em_max_toggle_rate` group's `index_1`. You can overwrite the values entered for the `em_lut_template` group's `index_2`, by entering a value for the `em_max_toggle_rate` group's `index_2`.

The following are the rules for the relationship between variables and indexes:

- If you have `variable_1`, you can have only `index_1`.
- If you have `variable_1` and `variable_2`, you can have `index_1` and `index_2`.
- The value you enter for `variable_1` (used for one-dimensional tables) is determined by how `index_1` is measured. The value you enter for `variable_2` (used for two-dimensional tables) is determined by how `index_2` is measured.

Example

```
index_1 ("0.0, 5.0, 20.0") ;
index_2 ("0.0, 1.0, 2.0") ;
```

electromigration Group

An electromigration group is defined in a `pin` group, as shown here:

```
library (name) {
    cell (name) {
        pin (name) {
            electromigration () {
                ... electromigration description ...
            }
        }
    }
}
```

}

Simple Attributes

```
related_pin : "name | name_list" /* path dependency */
related_bus_pins : "list of pins" /* list of pin names */
```

related_pin Simple Attribute

This attribute associates the `electromigration` group with a specific input pin. The input pin's input transition time is used as a variable in the electromigration lookup table.

If more than one input pin is specified in this attribute, the weighted input transition time of all input pins specified is used to index the electromigration table.

Syntax

```
related_pin : "name | name_list" ;
```

name | name_list

Name of input pin or pins.

Example

```
related_pin : "A B" ;
```

The pin or pins in the `related_pin` attribute denote the path dependency for the `electromigration` group. A particular `electromigration` group is accessed if the input pin or pins named in the `related_pin` attribute cause the corresponding output pin named in the `pin` group to toggle. All functionally related pins must be specified in a `related_pin` attribute if two-dimensional tables are being used.

Group Statements

```
em_max_toggle_rate (em_template_name) {}
```

These attributes and groups are described in the following sections.

related_bus_pins Simple Attribute

This attribute associates the `electromigration` group with the input pin or pins of a specific `bus` group. The input pin's input transition time is used as a variable in the electromigration lookup table.

If more than one input pin is specified in this attribute, the weighted input transition time of all input pins specified is used to index the electromigration table.

Syntax

```
related_bus_pins : "name1 [name2 name3 ... ]" ;
```

Example

```
related_bus_pins : "A" ;
```

The pin or pins in the `related_bus_pins` attribute denote the path dependency for the electromigration group. A particular electromigration group is accessed if the input pin or pins named in the `related_bus_pins` attribute cause the corresponding output pin named in the `pin` group to toggle. All functionally related pins must be specified in a `related_bus_pins` attribute if two-dimensional tables are being used.

`em_max_toggle_rate` Group

The `em_max_toggle_rate` group is a pin-level group that is defined within the electromigration `pin` group. The toggle rate is the number of toggles per unit of time where the unit of time is specified by the library-level `time_unit` attribute.

```
library (name) {
    cell (name) {
        pin (name) {
            electromigration () {
                em_max_toggle_rate(em_template_name) {
                    ... em_max_toggle_rate description ...
                }
            }
        }
    }
}
```

Simple Attribute

`current_type`

The `current_type` attribute is defined in the following section.

`current_type` Simple Attribute

The optional `current_type` attribute specifies the type of current for the `em_max_toggle_rate` lookup table. Valid values are `average`, `rms`, and `peak`.

Syntax

```
current_type: average | rms | peak ;
```

Example

```
current_type: average ;
```

Complex Attributes

```
index_1 ("float, ..., float") ; /*this attribute is
optional*/
index_2 ("float, ..., float") ; /*this attribute is
optional*/
values ("float, ..., float ") ;
```

These attributes are defined in the following sections.

Index_1 and Index_2 Complex Attributes

You can use the `index_1` optional attribute to specify the breakpoints of the first dimension of an electromigration table to characterize cells for electromigration within the library. You can use the `index_2` optional attribute to specify breakpoints of the second dimension of an electromigration table used to characterize cells for electromigration within the library.

You can overwrite the values entered for the `em_lut_template` group's `index_1`, by entering values for the `em_max_toggle_rate` group's `index_1`. You can overwrite the values entered for the `em_lut_template` group's `index_2`, by entering values for the `em_max_toggle_rate` group's `index_2`.

Syntax

```
index_1 ("float, ..., float") ; /*this attribute is  
optional*/  
index_2 ("float, ..., float") ; /*this attribute is  
optional*/
```

float

Floating-point numbers that identify the maximum toggle rate frequency.

Example

```
index_1 ("0.0, 5.0, 20.0") ;  
index_2 ("0.0, 1.0, 2.0") ;
```

values Complex Attribute

This complex attribute is used to specify the net's maximum toggle rates.

This attribute can be a list of `nindex_1` positive floating-point numbers if the table is one-dimensional.

This attribute can also be `nindex_1 X nindex_2` positive floating-point numbers if the table is two-dimensional, where `nindex_1` is the size of `index_1` and `nindex_2` is the size of `index_2` that is specified for these two indexes in the `em_max_toggle_rate` group or in the `em_lut_template` group.

Syntax

```
values("float, ..., float") ;
```

float

Floating-point numbers that identify the maximum toggle rate frequency.

Example (One-Dimensional Table)

```
values : ("1.5, 1.0, 0.5") ;
```

Example (Two-Dimensional Table)

```
values : ("2.0, 1.0, 0.5", "1.5, 0.75, 0.33", "1.0, 0.5, 0.15",) ;
```

em_temp_degradation_factor Simple Attribute

The `em_temp_degradation_factor` attribute specifies the electromigration exponential degradation factor.

Syntax

```
em_temp_degradation_factor : valuefloat ;
```

value

A floating-point number in centigrade units consistent with other temperature specifications throughout the library.

Example

```
em_temp_degradation_factor : 40.0 ;
```


13

Advanced Low-Power Modeling

Advanced low-power design methodologies such as multivoltage and multithreshold-CMOS require special library cells. These cells include power management attributes to drive implementation tools during library cell selection. This chapter describes the power and ground (PG) pin syntax and provides modeling examples of the special cells, such as the level-shifter, isolation, switch, and retention cells.

Note:

To model very deep submicron (VDSM) technology libraries, you must use the PG pin syntax.

This chapter includes the following sections:

- [Power and Ground \(PG\) Pins](#)
- [Silicon-on-Insulator \(SOI\) Cell Modeling](#)
- [Feedthrough Signal Pin Modeling](#)
- [Level-Shifter Cells in a Multivoltage Design](#)
- [Isolation Cell Modeling](#)
- [Switch Cell Modeling](#)
- [Retention Cell Modeling](#)
- [Always-On Cell Modeling](#)

- [Macro Cell Modeling](#)
- [Modeling Antenna Diodes](#)

Power and Ground (PG) Pins

The Library Compiler syntax supports power and ground (PG) library pins. A power pin is a current source pin, and a ground pin is a current sink pin. This section provides an overview of PG pins.

Partial PG Pin Cell Modeling

Partial PG pin cells are cells that have only power pins, only ground pins, or do not have power or ground PG pins. Partial PG pin cells have the following categories:

Table 13-1 Partial PG Pin Cell Categories

Partial PG Pin Cell	Description
Cells with one power pin and one ground pin	These cells are defined as having <ul style="list-style-type: none">• One or more primary power PG pins• One or more primary ground PG pins• Zero, or at least one, backup or internal power PG pins• Zero, or at least one, backup or internal ground PG pins• Zero, or at least one, nwell bias PG pins• Zero, or at least one, pwell bias PG pins
Cells with one power pin and no ground pins	These cells are defined as having <ul style="list-style-type: none">• One or more primary power PG pins• No primary ground PG pins• Zero, or at least one, backup or internal power PG pins• Zero, or at least one, backup or internal ground PG pins• Zero, or at least one, nwell bias PG pins• Zero, or at least one, pwell bias PG pins

Table 13-1 Partial PG Pin Cell Categories (Continued)

Partial PG Pin Cell	Description
Cells with no power pins and one ground pin	<p>These cells are defined as having</p> <ul style="list-style-type: none"> • No primary power PG pins • At least one primary ground PG pin • Zero, or at least one, backup or internal power PG pins • Zero, or at least one, backup or internal ground PG pins • Zero, or at least one, nwell bias PG pins • Zero, or at least one, pwell bias PG pins
Cells with no power pins or ground pins	<p>These cells are defined as having</p> <ul style="list-style-type: none"> • No primary power PG pins • No primary ground PG pins • No backup or internal power PG pin • No backup or internal ground PG pin • Zero, or at least one, nwell bias PG pins • Zero, or at least one, pwell bias PG pins

Special Partial PG Pin Cells

The following types of partial PG pin cells are acceptable with certain conditions. The tool issues a warning message and stores them in the .db file:

- ETM cells

ETM cells do not need a pair of PG pins. The tool generates a warning message for these cells, reporting the missing PG pin information. A cell is identified as an ETM cell if it has the `interface_timing` or `timing_model_type` attribute.

- Black box cells

A black box cell with no timing, noise, or power information does not need at least one power pin and one ground PG pin.

- Metal fills and antenna cells

Black box cells without functions do not need at least power pin and one ground pin.

- Cells without signal pins

Cells without signal pins do not need at least one power pin and one ground PG pin.

- Load cells

Cells without inout or output signal pins require either a power or a ground pin, but it is not necessary to have both.

- Tied-off cells and extensions

The following types of cells can be specified with one power pin and no ground PG pins:

- Cells with at least one inout or output pin with the `function` attribute set to 1.
- Cells with a pull-up signal, for example with the `driver_type` attribute set to `pull_up` or `pull_up_function`.
- Cells with a `resistive_1` signal, for example with the `driver_type` attribute set to `resistive_1` or `resistive_1_function`.

The following types of cells can be specified with no power pins and one ground PG pin:

- Cells with at least one inout or output pin with the `function` attribute set to 0.
- Cells with a pull-down signal, for example with the `driver_type` attribute set to `pull_down` or `pull_down_function`.
- Cells with a `resistive_0` signal, for example with the `driver_type` attribute set to `resistive_0` or `resistive_0_function`.

Supported Attributes

Cells with at least one power pin and no ground pins, cells with no power pins and at least one ground pin, and cells with no power pins or ground pins support the following attributes:

- To prevent a cell from being automatically inserted into the netlist, specify the `dont_touch` or the `dont_use` attribute. The `dont_touch` attribute set to `true` indicates that all instances of the cell must remain in the network. The `dont_use` attribute set to `true` indicates that a cell must not be added to a design during optimization.
- Cells with partial PG pins are reported using the following attributes:
 - `1p0g`
Reports cells with at least one power pin and no ground PG pins.
 - `0p1g`
Reports cells with no power pins and at least one ground PG pin.
 - `0p0g`
Reports cells with no power pins or ground PG pins.

For more information about library reports for partial PG pin cells, see the “Generating Library Reports” chapter in the *Library Quality Assurance System User Guide*.

Partial PG Pin Cell Example

[Example 13-1](#) shows a cell with only one primary ground pin.

Example 13-1 Partial PG Pin Cell With Only One Primary Ground Pin

```
cell (PULLDOWN) {
    pg_pin ( VSS ) {
        voltage_name : VSS;
        pg_type : primary_ground;
    }

    area : 1.0;
    dont_touch : true;
    dont_use : true;

    pin (X) {
        related_ground_pin : VSS;
        direction : output;
        function : "0";
        three_state : "!A";
        max_capacitance : 0.19;
        timing() {
            related_pin : "A";
            timing_type : three_state_enable;
            cell_rise(scalar) { values ( "0.1"); }
            rise_transition(scalar) { values ( "0.1"); }
            cell_fall(scalar) { values ( "0.1"); }
            fall_transition(scalar) { values ( "0.1"); }
        }
        timing() {
            related_pin : "A";
            timing_type : three_state_disable;
            cell_rise(scalar) { values ( "0.1"); }
            rise_transition(scalar) { values ( "0.1"); }
            cell_fall(scalar) { values ( "0.1"); }
            fall_transition(scalar) { values ( "0.1"); }
        }
    }
    pin (A) {
        related_ground_pin : VSS;
        direction : input;
        capacitance : 0.1;
        rise_capacitance : 0.1;
        rise_capacitance_range (0.1, 0.2);
        fall_capacitance : 0.1;
        fall_capacitance_range (0.1, 0.2);
    }
}
```

Partial PG Pin Cell Checks

The Library Compiler tool performs checks for partial PG pin cells and issues an error or warning message if certain conditions occur. For a detailed list of library checks for partial PG pin cells, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter of the *Library Quality Assurance System User Guide*.

PG Pin Syntax

The power and ground pin syntax for generic cells is as follows:

```
library(library_name) {
    ...
    voltage_map(voltage_name, voltage_value);
    voltage_map(voltage_name, voltage_value);
    ...
    operating_conditions(oc_name) {
        ...
        voltage : value;
        ...
    }
    ...
    default_operating_conditions : oc_name;
    cell(cell_name) {
        pg_pin (pg_pin_name_p1) {
            voltage_name : voltage_name_p1;
            pg_type : type_value;
        }
        pg_pin (pg_pin_name_g1) {
            voltage_name : voltage_name_g1;
            pg_type : type_value;
        }
        pg_pin (pg_pin_name_p2) {
            voltage_name : voltage_name_p2;
            pg_type : type_value;
        }
        pg_pin (pg_pin_name_g2) {
            voltage_name : voltage_name_g2;
            pg_type : type_value;
        }
        ...
        leakage_power() {
            related_pg_pin : pg_pin_name_p1;
            ...
        }
        ...
        pin (pin_name1) {
            direction : input | inout;
            related_power_pin : pg_pin_name_p1;
            related_ground_pin : pg_pin_name_g1;
            ...
        }
    }
}
```

```
}

...
pin (pin_name2) {
    direction : inout/output;
    power_down_function : (!pg_pin_name_p1 + !pg_pin_name_p2 + \
    !pg_pin_name_g1 + !pg_pin_name_g2) ;
    related_power_pin : pg_pin_namen_p2;
    related_ground_pin : pg_pin_namen_g2;
    internal_power() {
        related_pg_pin : pg_pin_name_p2;
        ...
    } /* end internal_power group */
    ...
}/* end pin group*/
...
}/* end cell group*/
...
}/* end library group*/
```

Library-Level Attributes

This section describes library-level attributes.

voltage_map Complex Attribute

The `voltage_map` attribute associates the voltage name with the relative voltage values. These voltage names are referenced by the `pg_pin` groups defined at the cell level. When specified in a library, this attribute identifies the library as a power and ground pin library. At least one voltage map in the library should have a value of 0, which becomes the reference value to which other voltage map values relate.

default_operating_conditions Simple Attribute

The `default_operating_conditions` attribute specifies the name of the default `operating_conditions` group in the library, which helps to identify the operating condition process, voltage, and temperature (PVT) points that are used during library characterization.

Cell-Level Attributes

This section describes cell-level attributes for `pg_pin` groups.

pg_pin Group

The pg_pin groups are used to represent the power and ground pins of a cell. Library cells can have multiple power and ground pins. The pg_pin groups are mandatory for each cell using the power and ground pin syntax, and a cell must have at least one primary_power power pin and at least one primary_ground ground pin.

is_pad Simple Attribute

The is_pad attribute identifies a pad pin on any I/O cell. The valid values are true and false. You can also specify the is_pad attribute on a PG pin. If the cell-level pad_cell attribute is specified on an I/O cell, you must set the is_pad attribute to true in either a pg_pin group or on a signal pin for that cell. Otherwise, the Library Compiler tool issues an error message. If the cell-level pad_cell attribute is specified on an I/O cell and there is no signal pin or PG pin in the pad cell, Library Compiler issues a warning message.

voltage_name Simple Attribute

The voltage_name string attribute is mandatory in all pg_pin groups except for a level-shifter cell not powered by the switching power domains. The voltage_name attribute specifies the associated voltage name of the power and ground pin defined using the voltage_map complex attribute at the library level.

Note:

To allow implementation tools to include a level-shifter cell not powered by the switching domains in any other power domain, the voltage value of the PG pin with the std_cell_main_rail attribute is ignored. So for this cell, the voltage_name attribute is optional in the pg_pin group that has the std_cell_main_rail attribute and not connected to the signal pins.

pg_type Simple Attribute

The pg_type attribute, optional in pg_pin groups, specifies the type of power and ground pin. The pg_type attribute can have the following values: primary_power, primary_ground, backup_power, backup_ground, internal_power, internal_ground, pwell, nwell, deepnwell and deeppwell.

The pg_type attribute also supports substrate-bias modeling. *Substrate bias* is a technique in which a *bias voltage* is varied on the substrate terminal of a CMOS device. This increases the threshold voltage, the voltage required by the transistor to switch, which helps reduce transistor power leakage. The pg_type attribute provides the pwell, nwell, deepnwell and deeppwell values to support substrate-bias modeling. The pwell and nwell values specify regular wells, and deepnwell and deeppwell specify isolation wells.

Table 13-2 describes the pg_type values.

Table 13-2 pg_type Values

Value	Description
primary_power	Specifies that pg_pin is a primary power source (the default). If the pg_type attribute is not specified, primary_power is the pg_type value.
primary_ground	Specifies that pg_pin is a primary ground source.
backup_power	Specifies that pg_pin is a backup (secondary) power source (for retention registers, always-on logic, and so on).
backup_ground	Specifies that pg_pin is a backup (secondary) ground source (for retention registers, always-on logic, and so on).
internal_power	Specifies that pg_pin is an internal power source for switch cells.
internal_ground	Specifies that pg_pin is an internal ground source for switch cells.
pwell	Specifies regular p-wells for substrate-bias modeling.
nwell	Specifies regular n-wells for substrate-bias modeling.
deepnwell	Specifies isolation n-wells for substrate-bias modeling.
deeppwell	Specifies isolation p-wells for substrate-bias modeling.

physical_connection Simple Attribute

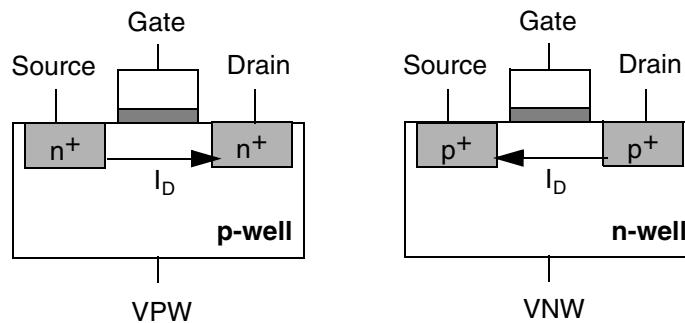
The physical_connection attribute can have the following values: device_layer and routing_pin. The device_layer value specifies that the bias connection is physically external to the cell. In this case, the library provides biasing tap cells that connect through the device layers. The routing_pin value specifies that the bias connection is inside a cell and is exported as a physical geometry and a routing pin. Macros with pin access generally use the routing_pin value if the cell has bias pins with geometry that is visible in the physical (FRAM) view.

related_bias_pin Attribute

The `related_bias_pin` attribute defines all bias pins associated with a power or ground pin within a cell. The `related_bias_pin` attribute is required only when the attribute is declared in a pin group but it does not specify a complete relationship between the bias pin and power and ground pin for a library cell.

The `related_bias_pin` attribute also defines all bias pins associated with a signal pin. To associate substrate-bias pins to signal pins, use the `related_bias_pin` attribute to specify one of the following `pg_type` values: `pwell`, `nwell`, `deeppwell`, `deepnwell`. [Figure 13-1](#) shows transistors with p-well and n-well substrate-bias pins.

Figure 13-1 Transistors With p-well and n-well Substrate-Bias Pins



The `related_bias_pin` attribute can be specified in the `pg_pin` group. If you define the `related_bias_pin` attribute for signal pins only and do not specify the attribute in the `pg_pin` group, the Library Compiler tool issues an error message. You do not need to specify bias PG pins again at the pin level because the `related_power_pin` and `related_ground_pin` attribute pairs that link a signal pin to primary, internal, or backup power and ground rails are already linked by the `related_bias_pin` attribute setting specified inside the `pg_pin` group.

user_pg_type Simple Attribute

The `user_pg_type` optional attribute allows you to customize the type of power and ground pin. It accepts any string value. The following example shows a `pg_pin` library with the `user_pg_type` attribute specified. The `user_pg_type` attribute must be specified with the `pg_type` attribute. If you do not specify `pg_type`, the power and ground pin value automatically defaults to `primary_power`.

```
pg_pin (pg_pin_name) {
    voltage_name : voltage_name;
    pg_type : primary_power | primary_ground |
               backup_power | backup_ground |
               internal_power | internal_ground;
    user_pg_type : user_pg_type_name;
}
```

Attributes Specifying Bias PG Pins With Insulated Substrate Wells

In power management designs, always-on bias PG pins are insulated from the shutdown bias PG pins. The substrate-bias PG pin of an always-on cell has an insulation layer around its pwell or nwell, or is placed in a separate region.

Use the following syntax to model the bias PG pins with insulated substrate wells.

```
pg_pin(bias_pgpin_name) {
    pg_type : pwell | nwell | deepnwell | deeppwell ;
    voltage_name : voltage_name ;
    is_insulated : true | false ;
    tied_to: pgpin_name ;
    ...
} /* end bias PG pin group */
```

The following attributes apply to PG pins with insulated substrate wells. You can use these attributes in both bulk CMOS and fully-depleted silicon-on-insulator (FDSOI) designs.

is_insulated Attribute

The `is_insulated` attribute specifies that the substrate-bias PG pin has an insulated well. The default is `false` meaning the bias PG pin substrate is not an insulated well. It is defined in a `pg_pin` group with the `pg_type` attribute set to `pwell`, `nwell`, `deeppwell`, or `deepnwell`.

tied_to Attribute

The optional `tied_to` attribute specifies the PG pin connected or tied to the substrate-bias PG pin.

Pin-Level Attributes

This section describes pin-level attributes.

power_down_function Attribute

The `power_down_function` string attribute specifies the Boolean condition under which the cell's output pin is switched off by the state of the power and ground pins (when the cell is in off mode due to the external power pin states). If `power_down_function` is evaluated as 1, X is assumed on the pin, meaning that the pin is assumed to be in an unknown state.

You specify the `power_down_function` attribute for combinational and sequential cells. For simple and complex sequential cells, `power_down_function` also determines the condition of the cell's internal state. Knowing the sequential cell's internal state is necessary for formal verification tools when they perform equivalence checking because the internal state is what is verified.

For more information about using the `power_down_function` attribute for sequential cells, see [Chapter 8, “Defining Sequential Cells”](#).

related_power_pin and related_ground_pin Attributes

The `related_power_pin` and `related_ground_pin` attributes associate a predefined power and ground pin with the signal pin, in which they are defined. This behavior only applies to standard cells. For special cells, explicitly specify this relationship.

If you do not specify the `related_power_pin` and `related_ground_pin` attribute values in a signal `pin` group in the `.lib` file, the tool automatically derives these attributes and assigns them with the following defaults:

- The first `pg_pin` group that has the `pg_type` attribute set to `primary_power` becomes the default for the `related_power_pin` attribute.
- The first `pg_pin` group that has the `pg_type` attribute set to `primary_ground` becomes the default for the `related_ground_pin` attribute.

The `pg_pin` groups are mandatory for each cell. Because a cell must have at least one `primary_power` and at least one `primary_ground` pin, a default `related_power_pin` and `related_ground_pin` always exists in any cell.

To disable the autoderivation of these attributes, specify:

```
lc_shell> set lc_disable_pg_pin_auto_association true
```

The default is `false`.

output_signal_level_low and output_signal_level_high Attributes

The `output_signal_level_low` and `output_signal_level_high` attributes can be defined at the pin level for the output pins and inout pins. The regular signal swings are derived for regular cells using the `related_power_pin` and `related_ground_pin` specifications.

input_signal_level_low and input_signal_level_high Attributes

The `input_signal_level_low` and `input_signal_level_high` attributes can be defined at the pin level for the input pins. The regular signal swings are derived for regular cells using the `related_power_pin` and `related_ground_pin` specifications.

related_pg_pin Attribute

The `related_pg_pin` attribute is used to associate a power and ground pin with leakage power and internal power tables. (The leakage power and internal power tables must be associated with the cell’s power and ground pins.)

In the absence of a `related_pg_pin` attribute, the `internal_power` and `leakage_power` specifications apply to the whole cell (cell-specific power specification).

Naming Conventions for Power and Ground Pins in Technology Libraries

The `pg_pin` names must match the names of the power and ground pins in the physical library exactly to correctly link the physical and logical views. For example, if the name of your power and ground pin in the physical view is VDD for power and VSS for ground, the same names should be specified in your logical library, as shown in the following example:

```
pg_pin(VDD) {  
...  
}  
/* and */  
pg_pin(VSS) {  
...  
}
```

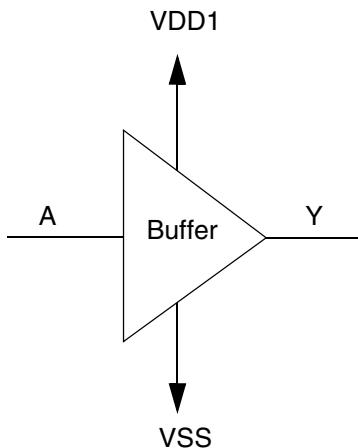
PG Pin Library Checks

The Library Compiler tool performs checks for standard cells that are based on PG pin syntax and issues an error or warning message if certain conditions occur. For a detailed list of library checks for PG pin libraries, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Standard Cell With One Power and Ground Pin Example

[Figure 13-2](#) shows a standard cell with a power and ground pin. The figure is followed by an example.

Figure 13-2 Standard Cell Buffer Schematic



```
library(standard_cell_library_example) {

    voltage_map(VDD, 1.0);
    voltage_map(VSS, 0.0);

    operating_conditions(XYZ) {
        voltage : 1.0;
        ...
    }
    default_operating_conditions : XYZ;

    cell(BUF) {

        pg_pin(VDD) {
            voltage_name : VDD;
            pg_type : primary_power;
        }
        pg_pin(VSS) {
            voltage_name : VSS;
            pg_type : primary_ground;
        }

        leakage_power() {
            related_pg_pin : VDD;
            when : "!A";
            value : 1.5;
        }
    }
}
```

```
pin(A) {
    related_power_pin : VDD;
    related_ground_pin : VSS;
}

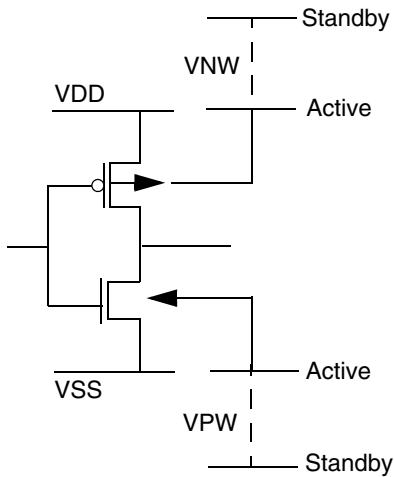
pin(Y) {
    direction : output;
    power_down_function : "!VDD + VSS";
    related_power_pin : VDD;
    related_ground_pin : VSS;

    timing() {
        related_pin : A;
        cell_rise(template) {
            ...
        }
        cell_fall(template) {
            ...
        }
        rise_transition(template) {
            ...
        }
        fall_transition(template) {
            ...
        }
    }
    internal_power() {
        related_pin : A;
        related_pg_pin : VDD;
        ...
    }
}/* end pin group*/
...
}/*end cell group*/
...
}/*end library group*/
```

Inverter With Substrate-Bias Pins Example

[Figure 13-3](#) shows an inverter with substrate-bias pins. The figure is followed by an example.

Figure 13-3 Inverter With Substrate-Bias Pins



```

library(low_power_cells) {

    delay_model : table_lookup;

    /* unit attributes */
    time_unit : "1ns";
    voltage_unit : "1V";
    current_unit : "1mA";
    pulling_resistance_unit : "1kohm";
    leakage_power_unit : "1pW";
    capacitive_load_unit (1.0, pf);

    voltage_map(VDD, 0.8); /* primary power */
    voltage_map(VSS, 0.0); /* primary ground */
    voltage_map(VNW, 0.8); /* bias power */
    voltage_map(VPW, 0.0); /* bias ground */

    /* operation conditions */
    operating_conditions(XYZ) {
        process: 1;
        temperature: 125;
        voltage: 0.8;
        tree_type: balanced_tree
    }
    default_operating_conditions : XYZ;
}

```

```
/* threshold definitions */
slew_lower_threshold_pct_fall : 30.0;
slew_upper_threshold_pct_fall : 70.0;
slew_lower_threshold_pct_rise : 30.0;
slew_upper_threshold_pct_rise : 70.0;
input_threshold_pct_fall     : 50.0;
input_threshold_pct_rise     : 50.0;
output_threshold_pct_fall    : 50.0;
output_threshold_pct_rise    : 50.0;

/* default attributes */
default_cell_leakage_power: 0.0;
default_fanout_load: 1.0;
default_output_pin_cap: 0.0;
default inout_pin_cap: 0.1;
default_input_pin_cap: 0.1;
default_max_transition: 1.0;

cell(std_cell_inv) {
    cell_footprint : inv;
    area : 1.0;
    pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
        related_bias_pin : "VNW";
    }
    pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
        related_bias_pin : "VPW";
    }
    pg_pin(VNW) {
        voltage_name : VNW;
        pg_type : nwell;
        physical_connection : device_layer;
    }
    pg_pin(VPW) {
        voltage_name : VPW;
        pg_type : pwell;
        physical_connection : device_layer;
    }

    pin(A) {
        direction : input;
        capacitance : 1.0;
        related_power_pin : VDD;
        related_ground_pin : VSS;
        related_bias_pin : "VPW VNW";
    }
    pin(Y) {
        direction : output;
        function : "A'";
    }
}
```

```

related_power_pin : VDD;
related_ground_pin : VSS;
related_bias_pin : "VPW VNW";
power_down_function : "!VDD + VSS + VNW' + VPW";
internal_power() {
    related_pg_pin : VDD;
    related_pin : "A";
    rise_power(scalar) { values ( "1.0"); }
    fall_power(scalar) { values ( "1.0"); }
}
timing() {
    related_pin : "A";
    timing_sense : positive_unate;
    cell_rise(scalar) { values ( "0.1"); }
    rise_transition(scalar) { values ( "0.1"); }
    cell_fall(scalar) { values ( "0.1"); }
    fall_transition(scalar) { values ( "0.1"); }
}
max_capacitance : 0.1;
}
cell_leakage_power : 1.0;
leakage_power() {
    when :"!A";
    value : 1.5;
}
leakage_power() {
    when :"A";
    value : 0.5;
}
}
}

```

Defining Power Data for Multiple-Rail Cells

The following example shows how to specify Liberty syntax to define power information for multiple-rail cells:

```

library(multi_rail_library) {
    delay_model : table_lookup;

    /* unit attributes */
    time_unit : "1ns";
    voltage_unit : "1V";
    current_unit : "1mA";
    pulling_resistance_unit : "1kohm";
    leakage_power_unit : "1pW";
    capacitive_load_unit (1.0,PF);

    /* power supply definition */
    voltage_map (VDD1, 1.0);
    voltage_map (VDD2, 2.0);
    voltage_map (VSS, 0.0);
}

```

```
operating_conditions(XYZ) {
    process : 1.0;
    voltage : 1.0;
    temperature : 25.0;
}
default_operating_conditions : XYZ;

cell (multi_rail_cell) {
    pg_pin (VDD1) {
        voltage_name : VDD1 ;
        pg_type : primary_power ;
    }

    pg_pin (VDD2) {
        voltage_name : VDD2 ;
        pg_type : primary_power ;
    }

    pg_pin (VSS) {
        voltage_name : VSS ;
        pg_type : primary_ground ;
    }
}

/* Specify the same number of when states for each rail. Otherwise,
The Library Compiler tool generates a parsing error. */

leakage_power() {
    related_pg_pin : "VDD1";
    when : "!A1 !A2" ;
    value : 1.5 ;
}
leakage_power() {
    related_pg_pin : "VDD1";
    when : "!A1 A2" ;
    value : 2.0 ;
}
leakage_power() {
    related_pg_pin : "VDD1";
    when : "A1 !A2" ;
    value : 3.0 ;
}
leakage_power() {
    related_pg_pin : "VDD2";
    when : "!A1 !A2" ;
    value : 3.5 ;
}
leakage_power() {
    related_pg_pin : "VDD2";
    when : "!A1 A2" ;
    value : 3.0 ;
}
leakage_power() {
    related_pg_pin : "VDD2";
    when : "A1 !A2" ;
    value : 4.0 ;
}
```

```

/* Default leakage power specification for each of the rails missing the
when states above */
leakage_power() {
    related_pg_pin : "VDD1";
    value : 3.0 ;
}
leakage_power() {
    related_pg_pin : "VDD2";
    value : 4.0 ;
}
area : 1.0 ;
pin(A1) {
    direction : input;
    capacitance : 0.1 ;
    related_power_pin : VDD1;
    related_ground_pin : VSS;
}
pin(A2) {
    direction : input;
    capacitance : 0.1 ;
    related_power_pin : VDD1;
    related_ground_pin : VSS;
}
pin(ZN) {
    direction : output;
    max_capacitance : 0.1;
    function : "(A1*A2)'";
    related_power_pin : VDD2;
    related_ground_pin : VSS;
}
timing() {
    related_pin : "A1"
    cell_rise( scalar ) {
        values("0.0");
    }
    rise_transition( scalar ) {
        values("0.0");
    }
    cell_fall( scalar ) {
        values("0.0");
    }
    fall_transition( scalar ) {
        values("0.0");
    }
}
internal_power() {
    related_pin : " A1 "
    related_pg_pin : "VDD1";
    rise_power( scalar ) {
        values("0.0");
    }
    fall_power( scalar ) {
        values("0.0");
    }
}
/* end of internal power */

internal_power() {
    related_pin : " A1 "
    related_pg_pin : "VDD2";
    rise_power( scalar ) {
        values("0.0");
    }
    fall_power( scalar ) {
        values("0.0");
    }
}

```

```
/* end of internal power */

timing() {
    related_pin : "A2"
    cell_rise( scalar ) {
        values("0.0");
    }
    rise_transition( scalar ) {
        values("0.0");
    }
    cell_fall( scalar ) {
        values("0.0");
    }
    fall_transition( scalar ) {
        values("0.0");
    }
}
internal_power() {
    related_pin : " A2 "
    related_pg_pin : "VDD1";
    rise_power( scalar ) {
        values("0.0");
    }
    fall_power( scalar ) {
        values("0.0");
    }
}
/* end of internal power */

internal_power() {
    related_pin : " A2 "
    related_pg_pin : "VDD2";
    rise_power( scalar ) {
        values("0.0");
    }
    fall_power( scalar ) {
        values("0.0");
    }
} /* end of internal power */
} /* end of pin group */
} /* end of cell */
```

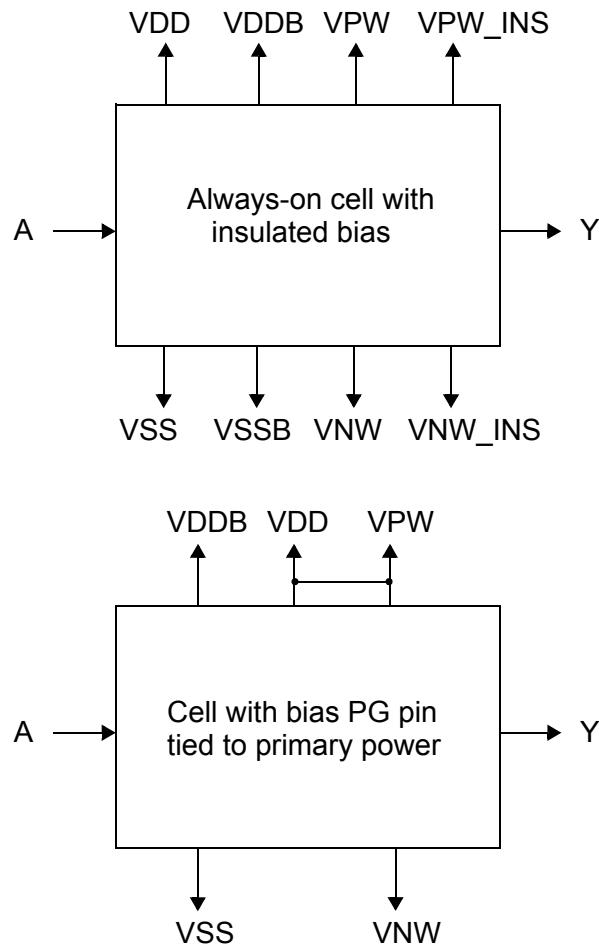
Insulated Bias Modeling

[Example 13-2](#) is a .lib description of a low-power library with the following two cells:

- An always-on cell with an insulated bias PG pin
- A cell where the bias PG pin is tied to the primary power supply

[Figure 13-4](#) shows the schematics of these cells.

Figure 13-4 An Always-On Cell With Insulated Bias and Cell Where the Bias PG pin is Tied to Primary Power



Example 13-2 A Low-Power Library Using the is_insulated and tied_to Attributes

```
library (mylib) {
    delay_model : table_lookup;

    /* library unit and slew attributes */
    voltage_map(VDD, 1.0);      /* Primary Power */
    voltage_map(VSS, 0.0);      /* Primary Ground */
    voltage_map(VDDB, 1.0);     /* Backup Power */
    voltage_map(VSSB, 0.0);     /* Backup Ground */
    voltage_map(VPW, 1.0);      /* nwell */
    voltage_map(VNW, 0.0);      /* pwell */
    voltage_map(VPW_INS, 1.0);  /* insulated nwell */
    voltage_map(VNW_INS, 0.0);  /* insulated pwell */
```

```
/* operation conditions */
nom_process      : 1.0;
nom_temperature : 25;
nom_voltage      : 1.0;
operating_conditions(XYZ) {
    process      : 1.0;
    temperature  : 25;
    voltage      : 1.0;
    tree_type    : balanced_tree
}
default_operating_conditions : XYZ;

/* AO cell with insulated bias pg_pins */
cell(AO_with_insulated_bias) {
    area : 1.0;
    ...
    pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
        related_bias_pin : VPW;
    }
    pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
        related_bias_pin : VNW;
    }
    pg_pin(VDDB) {
        voltage_name : VDDB;
        pg_type : backup_power;
        related_bias_pin : VPW_INS;
    }
    pg_pin(VSSB) {
        voltage_name : VSSB;
        pg_type : backup_ground;
        related_bias_pin : VNW_INS;
    }
    pg_pin(VPW) {
        voltage_name : VPW;
        pg_type : nwell;
        direction : inout;
        physical_connection : device_layer;
    }
    pg_pin(VNW) {
        voltage_name : VNW;
        pg_type : pwell;
        direction : inout;
        physical_connection : device_layer;
    }
    pg_pin(VNW_INS) {
        pg_type : pwell;
        voltage_name : VNW_INS;
        is_insulated : true;
    }
}
```

```

        tied_to : VSSB;
        direction : internal;
    }
    pg_pin(VPW_INS) {
        pg_type : nwell;
        voltage_name : VPW_INS;
        is_insulated : true;
        tied_to : VDDB;
        direction : internal;
    }
    pin(A) {
        direction : input;
        related_power_pin : VDDB;
        related_ground_pin : VSSB;
        ...
    }
    pin(Y) {
        direction : output;
        function : "A";
        related_power_pin : VDDB;
        related_ground_pin : VSSB;
        power_down_function : "!VDDB + !VPW_INS + VSSB + VNW_INS";
        ...
    } /* end pin group */
} /* end cell group */

/* cell with bias pg_pin tied to Primary Power */
cell(cell_with_bias_pg_pin_tied_to_power) {
    area : 1.0;
    ...
    pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
        related_bias_pin : VPW;
    }
    pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
        related_bias_pin : VNW;
    }
    pg_pin(VDDB) {
        voltage_name : VDDB;
        pg_type : backup_power;
        related_bias_pin : VPW_INS;
    }
    pg_pin(VPW) {
        voltage_name : VPW;
        pg_type : nwell;
        direction : inout;
        tied_to : VDD;
    }
    pg_pin(VNW) {
        voltage_name : VNW;
    }
}

```

```
    pg_type : pwell;
    direction : inout;
    physical_connection : device_layer;
}
pin(A) {
    direction : input;
    related_power_pin : VDDB;
    related_ground_pin : VSSB;
    ...
}
pin(Y) {
    direction : output;
    function : "A";
    related_power_pin : VDDB;
    related_ground_pin : VSSB;
    power_down_function : "!VDD + !VPW + VSSB + VNW";
    ...
} /* end pin group */
} /* end cell group */
} /* end library group */
```

Feedthrough Signal Pin Modeling

A feedthrough is created when two or more pins of a cell share the same physical or unbuffered logical net. Some feedthroughs have no electrical connections with the cell power pins and do not affect cell behavior.

Feedthrough pins can be of two types:

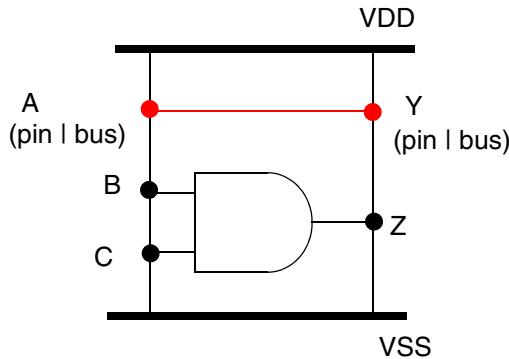
- Multipin, also known as logical feedthroughs or shorted pins
- Single-pin, also known as a pass-through, physical feedthrough, or floating pin

Feedthrough pins can also be part of buses.

Multipin Feedthroughs

[Figure 13-5](#) is a schematic of a two-pin feedthrough cell with feedthrough pins, A and Y. Pins A and Y appear in all the model views (such as Verilog) as two separate external pins.

Figure 13-5 A Two-Pin Feedthrough



Multipin Feedthrough Syntax

```

library (library_name) {
    cell(cell_name) {
        ...
        short(pin_name, pin_name, ...);
        ...
        pin(pin_name) {
            direction : inout;
        }
        ...
    } /* end cell group */
}
/* end library group */
    
```

To model multipin feedthroughs as shown in [Figure 13-5](#), use the `short` attribute in the `cell` group.

short Attribute

The `short` complex attribute lists the shorted ports or pins that are connected by metal or polytrace. You must list at least two pin names.

To model multiple sets of feedthrough pins in a cell, specify multiple `short` attributes. For example, if pins a and b are shorted to create a feedthrough, and pins c and d are shorted to create another feedthrough, use:

```

short (a,b);
short (c,d);
    
```

However, if pin a is shorted with pin b, and pin a is also shorted with pin c, that is, pins a, b, and c are shorted, use:

```

short (a,b,c);
    
```

This is equivalent to:

```
short (a,b);
short (a,c);
```

The tool recognizes a cell where the `short` attribute is defined as a feedthrough cell, and automatically marks the cell with the `is_feed_through_cell` attribute set to `true`.

The tool also recognizes a pin specified in the `short` attribute as a feedthrough pin, and automatically marks the pin with the `is_feed_through_pin` attribute set to `true`.

Do not specify the `related_power_pin` or the `related_ground_pin` attribute for a feedthrough pin, that is, a pin listed in the `short` attribute. Otherwise, the Library Compiler tool generates an error message.

Note:

The Power Compiler tool can connect the feedthrough pin to a desired power domain.

[Example 13-3](#) is a library model of the cell shown in [Figure 13-5](#).

Example 13-3 A Two-Pin Feedthrough Cell Model

```
library(mylib) {
  cell(mycell) {
    ...
    /* signal pins A and Y are feedthrough pins */
    short(A, Y);
    ...
    pin(A) { /* Do not specify related_power_pin or related_ground_pin */
      /* set pin direction based on actual cell characteristics */
      direction : inout;
      ...
    }
    pin(Y) { /* Do not specify related_power_pin or related_ground_pin */
      direction : inout;
      ...
    } /* end pin group */
    pin (B C) {
      related_power_pin : "VDD";
      related_ground_pin : "VSS";
      direction : "input";
      ...
    } /* end pin group */
    pin (Z) {
      related_power_pin : "VDD";
      related_ground_pin : "VSS";
      direction : "output";
      power_down_function : "!VDD + VSS";
      function : "B * C";
      timing () {
        related_pin : "B C";
        ...
      }
      ...
    } /* end pin group */
  }
}
```

```

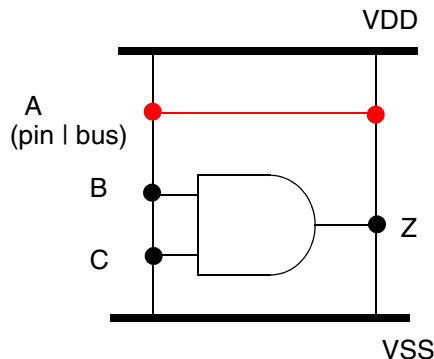
} /* end cell group */
...
} /* end library group */

```

Single-Pin Feedthrough

[Figure 13-6](#) is a schematic of a single-pin feedthrough cell, with feedthrough pin A. In all the model views, only one external pin, bus, or bundle, A, is declared even though the signal can establish one or more physical connections with other cells.

Figure 13-6 Single-Pin Feedthrough



Single-Pin Feedthrough Syntax

```

library (library_name) {
    cell(cell_name) {
        ...
        pin (single_pin_feedthrough_name) {
            direction : inout;
        } /* end pin group */
        ...
    } /* end cell group */
} /* end library group */

```

To model a single-pin feedthrough, as shown in [Figure 13-6](#):

- Specify the feedthrough pin as a floating or unused signal pin without parasitics.
- Do not specify any `related_power_pin` and `related_ground_pin` associations for these pins.
- Set the Tcl `lc_disable_pg_pin_auto_association` variable to `true` before running the `read_lib` command. This prevents the Library Compiler tool from automatically deriving the `related_power_pin` and `related_ground_pin` attributes for cells with single-pin feedthrough signal pins.

By default, the variable is `false` and the `related_power_pin` and `related_ground_pin` attributes are automatically derived during the `read_lib` run.

[Example 13-4](#) is a library model of the cell shown in [Figure 13-6](#).

Example 13-4 A Single-Pin Feedthrough Model

```
library(mylib) {
    cell (mycell) {
        ...
        pg_pin (VDD) {
            voltage_name : "VDD";
            pg_type : "primary_power";
        }
        pg_pin (VSS) {
            voltage_name : "VSS";
            pg_type : "primary_ground";
        }
        pin (B C) {
            related_power_pin : "VDD";
            related_ground_pin : "VSS";
            direction : "input";
            ...
        }
        pin (A) /* Do not specify related_power_pin or related_ground_pin */
            /* set pin direction based on actual cell characteristics */
            direction : "inout";
            ...
        }
        pin (Z) {
            related_power_pin : "VDD";
            related_ground_pin : "VSS";
            direction : "output";
            function : "B * C";
            timing () {
                related_pin : "B C";
                ...
            }
            ...
        }
        ...
    } /* end cell group*/
    ...
} /* end library group */
```

Silicon-on-Insulator (SOI) Cell Modeling

Silicon-on-insulator (SOI) devices are fabricated on a silicon layer that rests upon an insulating layer on the substrate. The presence of the insulating layer makes the drain and source junctions shallow, with small capacitances. Therefore, the SOI devices have better electrical characteristics resulting in improved switching speed and reduced power dissipation.

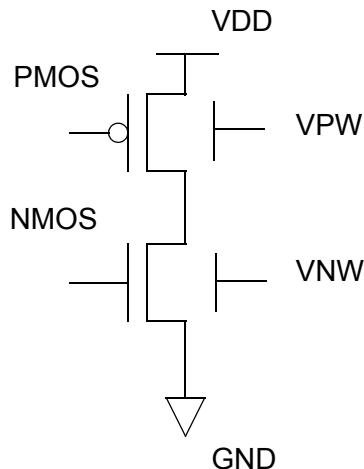
The isolation of the silicon layer from the substrate improves the operating speed, reduces leakage to the substrate, and reduces the number of latch-ups. It is easy to closely pack these inherently insulated structures for high device densities. Further, the SOI fabrication process requires only minor changes to the bulk-CMOS process flow.

Depending on the thickness of the silicon layer, an SOI cell is of two types:

- Fully depleted SOI (FDSOI) cell, where the channel depletion width is greater than the thickness of the silicon layer. Therefore, the channel is fully depleted.
- Partially depleted SOI (PDSOI) cell, where the channel depletion width is less than the thickness of the silicon layer. Therefore, the channel is partially depleted.

[Figure 13-7](#) shows a typical SOI cell schematic.

Figure 13-7 A Typical SOI Cell Schematic



The modeling syntax of the SOI cell is:

```

library(library_name) {
    ...
    is_soi: true | false;
    ...
    cell(cell_name) {
        ...
    }
}
    
```

```

    is_soi: true | false;
    ...
} /* End cell group */
} /* End Library group */

```

Table 13-3 shows the differences in modeling substrate-bias pins between a bulk-CMOS and the SOI cell.

Table 13-3 Differences in Substrate-Bias Pin Modeling Between Bulk-CMOS and SOI Cell

Bulk-CMOS	SOI
For a substrate-bias pin associated with a primary power pin, the pg_type attribute must be nwell.	For a substrate-bias pin associated with a primary power pin, the pg_type attribute can be pwell or nwell.
For a substrate-bias pin associated with a primary ground pin, the pg_type attribute must be pwell.	For a substrate-bias pin associated with a primary ground pin, the pg_type attribute can be pwell or nwell.

Example 13-5 shows a typical SOI library and cell model.

Example 13-5 An SOI Library and Cell Description

```

library(SOI) {
    delay_model : table_lookup;
    /* unit attributes */

    ...
    is_soi : true;
    ...
    /* operation conditions */
    ...
    /* threshold definitions */
    ...
    /* default attributes */
    ...
    cell(std_cell_inv) {
        is_soi : true;
        cell_footprint : inv;
        area : 1.0;
        pg_pin(vdd) {
            voltage_name : vdd;
            pg_type : primary_power;
            related_bias_pin : "vpw";
        }
    }
}

```

Library-Level Attribute

This section describes a library-level attribute of SOI cells.

is_soi Attribute

The `is_soi` attribute specifies that all the cells in a library are SOI cells. The default is `false`, which means that the library cells are bulk-CMOS cells.

If the `is_soi` attribute is specified at both the library and cell levels, the cell-level value overrides the library-level value.

Cell-Level Attribute

This section describes a cell-level attribute of SOI cells.

is_soi Attribute

The `is_soi` attribute specifies that the cell is an SOI cell. The default is `false`, which means that the cell is a bulk-CMOS cell.

SOI Cell Checks

The Library Compiler tool checks for conditions of SOI cells and issues an error or warning message if certain conditions occur. For a detailed list of these library checks, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Level-Shifter Cells in a Multivoltage Design

Level-shifter cells (or buffer-type level-shifter cells) and isolation cells are used to connect the netlist in different power domains, to meet design constraints. In multivoltage and shut-down designs, both level shifting and isolation are required. An enable level-shifter cell fulfills both these requirements because it can function as an isolation or a level-shifter cell.

Implementation tools need the following information about level-shifter cells from the cell library:

- Which power and ground pin of the level-shifter cell is used for voltage boundary hookup during its insertion. This information allows the optimization tools to determine on which side of the voltage boundary a particular level-shifter cell is allowed.

- Which voltage conversions the particular level-shifter cell can handle. Specifically, does the level-shifter cell work for conversion from high voltage to low voltage (HL), from low voltage to high voltage (LH), or both (HL_LH)?
- What the input and output voltage ranges are for a level-shifter cell under all operating conditions.

Operating Voltages

In a multivoltage design, each design instance can operate at its specified operating voltage. Therefore, each voltage must correspond to one or more logical hierarchies. All cells in a hierarchy operate at the same voltage except for level shifters.

The operating voltages are annotated on the top design, design instances, or hierarchical ports through PVT operating conditions.

Level Shifter Functionality

A level shifter functions like a buffer, except that the input pin and output pin voltages are different. These cells are necessary in a multivoltage design because the nets connecting pins at two different operating voltages can cause a design violation. Level shifters provide these nets with the needed voltage adjustments.

A level shifter has two components:

- Two power supplies
- Specified input and output voltages

The functionality of level shifters includes

- Identifying nets in the design that need voltage adjustments
- Analyzing the target library for the availability of level shifters
- Ripping the net and instantiating level shifters where appropriate

Basic Level-Shifter Syntax

The basic syntax for level-shifter cells is as follows:

```
cell(level_shifter) {  
    is_level_shifter : true ;  
    level_shifter_type : HL | LH | HL_LH ;  
    input_voltage_range ("float, float");  
    output_voltage_range ("float, float");
```

```

...
pg_pin(pg_pin_name_P) {
    pg_type : primary_power;
    std_cell_main_rail : true;
    ...
}
pg_pin(pg_pin_name_G) {
    pg_type : primary_ground;
    ...
}

pin (data) {
    direction : input;
    input_signal_level : "voltage_rail_name";
    input_voltage_range ("float , float");
    level_shifter_data_pin : true ;
    ...
}/* End pin group */

pin (enable) {
    direction : input;
    input_voltage_range ("float , float");
    level_shifter_enable_pin : true ;
    ...
}/* End pin group */

pin (output) {
    direction : output;
    output_voltage_range ("float , float");
    power_down_function : (!pg_pin_name_P + pg_pin_name_G);
    ...
}/* End pin group */

...
}/* End Cell group */

```

Cell-Level Attributes

This section describes cell-level attributes for level-shifter cells.

is_level_shifter Attribute

The `is_level_shifter` simple attribute identifies a cell as a level shifter. The valid values of this attribute are `true` or `false`. If not specified, the default is `false`, meaning that the cell is a standard cell.

level_shifter_type Attribute

The `level_shifter_type` complex attribute specifies the supported voltage conversion type. The valid values are

LH

Low to high

HL

High to low

HL_LH

High to low and low to high

The `level_shifter_type` attribute is optional. The default is HL_LH.

input_voltage_range Attribute

The `input_voltage_range` attribute specifies the allowed voltage range of the level-shifter input pin and the voltage range for all input pins of the cell under all possible operating conditions (defined across multiple libraries). The attribute defines two floating values: the first is the lower bound, and the second is the upper bound.

The `input_voltage_range` syntax differs from the pin-level `input_signal_level_low` and `input_signal_level_high` syntax in the following ways:

- The `input_signal_level_low` and `input_signal_level_high` attributes are defined on the input pins under one operating condition (the default operating condition of the library).
- The `input_signal_level_low` and `input_signal_level_high` attributes specify the partial voltage swing of an input pin. Use these attributes to specify partial swings rather than the full rail-to-rail swing. The `input_voltage_range` attribute is not related to the voltage swing.

Note:

The `input_voltage_range` and `output_voltage_range` attributes must be defined together. Otherwise, the Library Compiler tool issues an error message.

output_voltage_range Attribute

The `output_voltage_range` attribute is similar to the `input_voltage_range` attribute except that it specifies the allowed voltage range of the level shifter for the output pin instead of the input pin.

The `output_voltage_range` syntax differs from the pin-level `output_signal_level_low` and `output_signal_level_high` syntax in the following ways:

- The `output_signal_level_low` and `output_signal_level_high` attributes are defined on the output pins under one operating condition (the default operating condition of the library).
- The `output_signal_level_low` and `output_signal_level_high` attributes specify the partial voltage swing of an output pin. Use these attributes to specify partial swings rather than the full rail-to-rail swing. The `output_voltage_range` attribute is not related to the voltage swing.

Note:

The `input_voltage_range` and `output_voltage_range` attributes must be defined together. Otherwise, the Library Compiler tool issues an error message.

Pin-Level Attributes

This section describes pin-level attributes for level-shifter cells.

Note:

The `level_shifter_data_pin`, `level_shifter_enable_pin`, `input_voltage_range`, `output_voltage_range`, and the output clamp function (see “[Clamping Enable Level-Shifter Cell Outputs](#)” on page 13-38) attributes also apply to low-power complex macro cell models. For more information, see “[Low Power Macro Cell Modeling](#)” on page 13-144.

std_cell_main_rail Attribute

The `std_cell_main_rail` attribute is defined in a `primary_power` power pin. When the attribute is set to `true`, the `pg_pin` is used to determine which power pin is the main rail in the cell.

level_shifter_data_pin Attribute

The `level_shifter_data_pin` attribute identifies a data pin of a level-shifter cell. The default is `false`, meaning that the pin is a regular signal pin.

level_shifter_enable_pin Attribute

The `level_shifter_enable_pin` attribute identifies an enable pin of a level-shifter cell. The default is `false`, meaning that the pin is a regular signal pin.

input_voltage_range and output_voltage_range Attributes

The `input_voltage_range` and `output_voltage_range` attributes specify the allowed voltage ranges of the input or an output pin of the level-shifter cell. The attributes define two floating values where the first value is the lower bound and the second value is the upper bound.

Note:

The pin-level attribute specifications always override the cell-level specifications.

input_signal_level Attribute

The `input_signal_level` attribute is defined at the pin level and is used to specify which signal is driving the input pin. The attribute defines special overdrive cells that do not have a physical relationship with the power and ground on input pins.

If the `input_signal_level`, `related_power_pin`, and `related_ground_pin` attributes are defined on any input pin, the full voltage swing derived from the `input_signal_level` attribute takes precedence over the voltage swing derived from the `related_power_pin` and `related_ground_pin` attributes during timing calculations.

power_down_function Attribute

The `power_down_function` attribute identifies the Boolean condition under which the cell's signal output pin is switched off (when the cell is in the off mode due to the external power pin states). If the `power_down_function` attribute is set to 1, then X is assumed on the pin.

alive_during_power_up Attribute

The optional `alive_during_power_up` attribute specifies an active data pin that is powered by a more always-on supply. The default is `false`. If set to `true`, it indicates that the data pin is active while its related power pin is active.

You can specify this attribute only in a `pin` group where the `isolation_cell_data_pin` or the `level_shifter_data_pin` attribute is set to `true`.

Clamping Enable Level-Shifter Cell Outputs

In enable level-shifter (ELS) cells with multiple enable pins, clamping the outputs might be required to maintain them at particular logic levels, such as 0, 1, z, or a previously-latched value.

The clamp function modeling syntax for an enable level-shifter cell and its pins is as follows:

```
cell (cell_name) {
    is_level_shifter : true;
```

```

...
pin(input_pin) {
    direction : input;
    level_shifter_data_pin : true;
    ...
}
pin(input_pin) {
    direction : input;
    level_shifter_enable_pin : true;
    ...
}
pin(input_pin) {
    direction : input;
    level_shifter_enable_pin : true;
    ...
}
pin(output_pin_name) {
    direction : output;
    clamp_0_function : "Boolean expression" ;
    clamp_1_function : "Boolean expression" ;
    clamp_z_function : "Boolean expression" ;
    clamp_latch_function : "Boolean expression" ;
    illegal_clamp_condition : "Boolean expression" ;
    ...
}
...
}

```

Pin-Level Attributes

This section describes the pin-level clamp function attributes to clamp the output pins of an enable level-shifter cell.

clamp_0_function Attribute

The `clamp_0_function` attribute specifies the input condition for the enable pins of an enable level-shifter cell when the output clamps to 0.

clamp_1_function Attribute

The `clamp_1_function` attribute specifies the input condition for the enable pins of an enable level-shifter cell when the output clamps to 1.

clamp_z_function Attribute

The `clamp_z_function` attribute specifies the input condition for the enable pins of an enable level-shifter cell when the output clamps to z.

clamp_latch_function Attribute

The `clamp_latch_function` attribute specifies the input condition for the enable pins of an enable level-shifter cell when the output clamps to the previously latched value.

illegal_clamp_condition Attribute

The `illegal_clamp_condition` attribute specifies the invalid condition for the enable pins of an enable level-shifter cell. If the `illegal_clamp_condition` attribute is not specified, the invalid condition does not exist.

Note:

All the input pins used in the Boolean expressions of the `clamp_0_function`, `clamp_1_function`, `clamp_z_function`, and `clamp_latch_function` attributes must be enable pins with the `level_shifter_enable_pin` attribute set to `true`. The Boolean expressions of the `clamp_0_function`, `clamp_1_function`, `clamp_z_function`, `clamp_latch_function`, and `illegal_clamp_condition` attributes must be mutually exclusive.

Level-Shifter Cell With PG Pin Not Connected to Switching Domains

The Library Compiler tool supports level-shifter cells that are not powered by the supply voltages of the power domains involved in level-shifting. For example, a level-shifter cell that shifts voltage levels from one power domain supply, VDD1, to another power domain supply, VDD2, and is powered by a third power domain supply, VDD3.

This level-shifter cell model removes the limitation of placing the level-shifter cell in one of the level-shifting or switching power domains during implementation. The cell can now be part of any other power domain. This enables level-shifting of always-on signals between shutdown power domains.

The level-shifter cell has an additional power PG pin with the `std_cell_main_rail` attribute set to `true`, which is not connected to the signal pins of the level-shifter cell and is not specified as the value of the `related_power_pin` attribute in any of the signal pins. The `voltage_name` attribute is optional in this `pg_pin` group. To directly identify this PG pin, the Library Compiler tool automatically marks the PG pin with the `is_unconnected_pg_pin` attribute set to `true` during library compilation.

Syntax

The following syntax is the Liberty model of a multirail level-shifter cell that has a PG pin with the `std_cell_main_rail` attribute and is not connected to the switching power domains:

```
cell(level_shifter) {  
    is_level_shifter : true ;  
}
```

```

level_shifter_type : HL | LH | HL_LH ;
...
pg_pin(pg_pin_name_dummy) { /* Unconnected std_cell_main_rail pg_pin */
    pg_type : primary_power;
    std_cell_main_rail : true;
    voltage_name : ...;
    ...
}
pg_pin(pg_pin_name_P1) {
    pg_type : primary_power;
    voltage_name : ...;
    ...
}
pg_pin(pg_pin_name_P2) {
    pg_type : primary_power;
    voltage_name : ...;
    ...
}
pg_pin(pg_pin_name_G) {
    pg_type : primary_ground;
    voltage_name :
    ...
}
pin (input) {
    direction : input;
    related_power_pin : "pg_pin_name_P1";
    related_ground_pin : "pg_pin_name_G";
    ...
}
pin (output) {
    direction : output;
    related_power_pin : "pg_pin_name_P2";
    related_ground_pin : "pg_pin_name_G";
    ...
} /* End pin group */
...
} /* End Cell group */

```

The following syntax shows the Liberty model of an overdrive level-shifter cell that has the PG pin with the `std_cell_main_rail` attribute and is not connected to the switching power domains:

```

cell(level_shifter) {
    is_level_shifter : true ;
    level_shifter_type : HL | LH | HL_LH ;
    ...
    pg_pin(pg_pin_name_dummy) { /* dummy unconnected std_cell_main_rail
        pg_pin */
        pg_type : primary_power;
        std_cell_main_rail : true;
        ...
    }
    pg_pin(pg_pin_name_P) {

```

```
    pg_type : primary_power;
    ...
}
pg_pin(pg_pin_name_G) {
    pg_type : primary_ground;
    ...
}
pin (input) {
    direction : input;
    input_signal_level : string; /* voltage map name string*/
    level_shifter_data_pin : true;
    related_power_pin : "pg_pin_name_P";
    related_ground_pin : "pg_pin_name_G";
    ...
} /* End pin group */
pin (output) {
    direction : output;
    related_power_pin : "pg_pin_name_P";
    related_ground_pin : "pg_pin_name_G";
    ...
} /* End pin group */
...
} /* End Cell group */
```

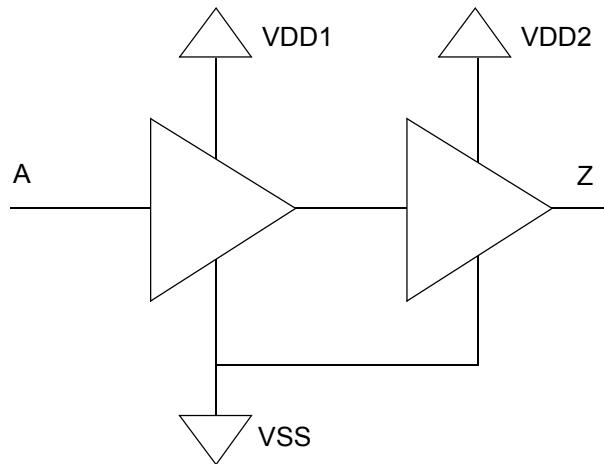
Level Shifter Modeling Examples

The following sections provide examples for modeling a simple buffer type low-to-high level-shifter cell, a simple buffer type high-to-low level-shifter cell, an overdrive high-to-low level-shifter cell, a level-shifter cell with virtual bias pins, and enable level-shifter cells.

Simple Buffer Type Low-to-High Level Shifter

[Figure 13-8](#) shows a simple buffer type low-to-high level-shifter cell modeled using the power and ground pin syntax and level-shifter attributes. The figure is followed by an example.

Figure 13-8 Buffer Type Low-to-High Level-Shifter Cell



```

library(level_shifter_cell_library_example) {
    voltage_map(VDD1, 0.8);
    voltage_map(VDD2, 1.2);
    voltage_map(VSS, 0.0);
    operating_conditions(XYZ) {
        process : 1.0;
        voltage : 3.0;
        temperature : 25.0;
    }
    default_operating_conditions : XYZ;

    cell(Buffer_Type_LH_Level_shifter) {
        is_level_shifter : true;
        level_shifter_type : LH ;

        pg_pin(VDD1) {
            voltage_name : VDD1;
            pg_type : primary_power;
            std_cell_main_rail : true;
        }
        pg_pin(VDD2) {
            voltage_name : VDD2;
            pg_type : primary_power;
        }
        pg_pin(VSS) {
            voltage_name : VSS;
            pg_type : primary_ground;
        }

        leakage_power() {
            when : "!A";
            value : 1.5;
            related_pg_pin : VDD1;
        }
    }
}
  
```

```

leakage_power() {
    when : "!A";
    value : 2.7;
    related_pg_pin : VDD2;
}

pin(A) {
    direction : input;
    related_power_pin : VDD1;
    related_ground_pin : VSS;
    input_voltage_range ( 0.7 , 0.9);
}

pin(Z) {
    direction : output;
    related_power_pin : VDD2;
    related_ground_pin : VSS;
    function : "A";
    power_down_function : "!VDD1 + !VDD2 + VSS";
    output_voltage_range (1.1 , 1.3);

    timing() {
        related_pin : A;
        cell_rise(template) {
            ...
        }
        cell_fall(template) {
            ...
        }
        rise_transition(template) {
            ...
        }
        fall_transition(template) {
            ...
        }
    }

    internal_power() {
        related_pin : A;
        related_pg_pin : VDD1;
        ...
    }
    internal_power() {
        related_pin : A;
        related_pg_pin : VDD2;
        ...
    }
}

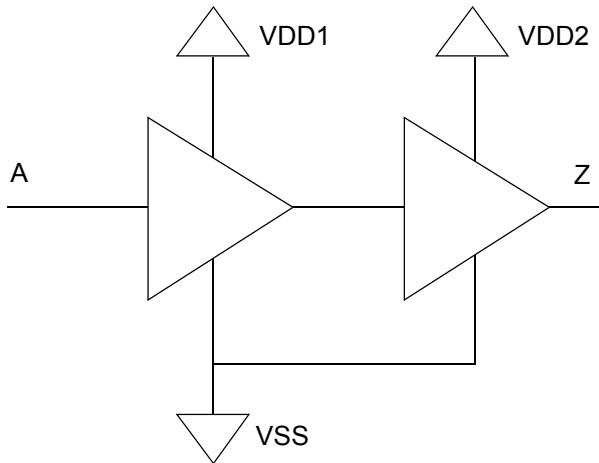
} /* end pin group*/
...
} /*end cell group*/
...
} /*end library group*/

```

Simple Buffer Type High-to-Low Level Shifter

[Figure 13-9](#) shows a simple buffer type high-to-low level-shifter cell. The cell is modeled using the power and ground pin syntax and level-shifter attributes. Shifting the signal level from high to low voltage can be useful for timing accuracy. When you do this, the level-shifter cell receives a higher voltage signal as its input, which is characterized in the delay tables of the cell description.

Figure 13-9 Buffer Type High-to-Low Level-Shifter Cell



```

library(level_shifter_cell_library_example) {
    voltage_map(VDD1, 1.2);
    voltage_map(VDD2, 0.8);
    voltage_map(VSS, 0.0);
    operating_conditions(XYZ) {
        process : 1.0;
        voltage : 3.0;
        temperature : 25.0;
    }
    default_operating_conditions : XYZ;

    cell(Buffer_Type_HL_Level_shifter) {
        is_level_shifter : true;
        level_shifter_type : HL ;

        pg_pin(VDD1) {
            voltage_name : VDD1;
            pg_type : primary_power;
            std_cell_main_rail : true;
        }
        pg_pin(VDD2) {
            voltage_name : VDD2;
            pg_type : primary_power;
        }
        pg_pin(VSS) {
  
```

```
voltage_name : VSS;
pg_type : primary_ground;
}

leakage_power() {
    when : "!A";
    value : 1.5;
    related_pg_pin : VDD1;
}
leakage_power() {
    when : "!A";
    value : 2.7;
    related_pg_pin : VDD2;
}

pin(A) {
    direction : input;
    related_power_pin : VDD1;
    related_ground_pin : VSS;
    input_voltage_range ( 0.7 , 0.9 );
}

pin(Z) {
    direction : output;
    related_power_pin : VDD2;
    related_ground_pin : VSS;
    function : "A";
    power_down_function : "!VDD1 + !VDD2 + VSS";
    output_voltage_range (1.1 , 1.3);

    timing() {
        related_pin : A;
        cell_rise(template) {
            ...
        }
        cell_fall(template) {
            ...
        }
        rise_transition(template) {
            ...
        }
        fall_transition(template) {
            ...
        }
    }

    internal_power() {
        related_pin : A;
        related_pg_pin : VDD1;
        ...
    }
    internal_power() {
        related_pin : A;
```

```

    related_pg_pin : VDD2;
    ...
}

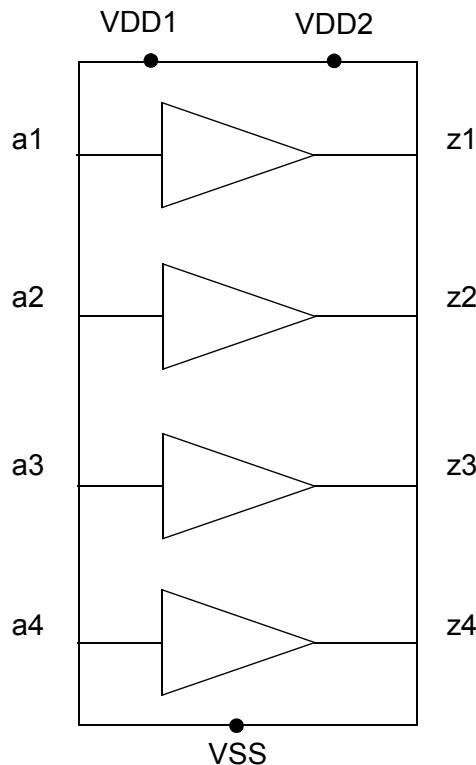
} /* end pin group*/
...
} /*end cell group*/
...
} /*end library group*/

```

Multibit Level-Shifter Cell

[Figure 13-10](#) shows a multibit buffer-type level-shifter cell with four data input and four data output pins. Each of the outputs is a linear function of the respective input. The input data pins are powered by the PG pin, VDD1, and the output data pins are powered by the PG pin, VDD2. The example following the figure is a model of the level-shifter cell.

Figure 13-10 4-bit Level-Shifter Cell



```

cell (4_bit_ls) {
    is_level_shifter : true;
    pg_pin (VDD1) {
        pg_type : primary_power;
        voltage_name : VDD1;
    }
}

```

```
        }
    pg_pin (VDD2) {
        pg_type : primary_power;
        voltage_name : VDD2;
    }
    pg_pin (VSS) {
        pg_type : primary_ground;
        voltage_name : VSS;
    }
    ...
    pin (a1) {
        level_shifter_data_pin : true;
        related_power_pin : VDD1;
        related_ground_pin : VSS;
        ...
    }
    pin (a2) {
        level_shifter_data_pin : true;
        related_power_pin : VDD1;
        related_ground_pin : VSS;
        ...
    }
    pin (a3) {
        level_shifter_data_pin : true;
        related_power_pin : VDD1;
        related_ground_pin : VSS;
        ...
    }
    pin (a4) {
        level_shifter_data_pin : true;
        related_power_pin : VDD1;
        related_ground_pin : VSS;
        ...
    }
    pin (z1) {
        direction : output;
        related_power_pin : VDD2;
        related_ground_pin : VSS;
        power_down_function : !VDD1+!VDD2+VSS;
        function : a1;
        ...
    }
    pin (z2) {
        direction : output;
        related_power_pin : VDD2;
        related_ground_pin : VSS;
        power_down_function : !VDD1+!VDD2+VSS;
        function : a2;
        ...
    }
    pin (z3) {
        direction : output;
        related_power_pin : VDD2;
```

```

related_ground_pin : VSS;
power_down_function : !VDD1+!VDD2+VSS;
function : a3;
...
}
pin (z4) {
  direction : output;
  related_power_pin : VDD2;
  related_ground_pin : VSS;
  power_down_function : !VDD1+!VDD2+VSS;
  function : a4;
}
} /* end cell */

```

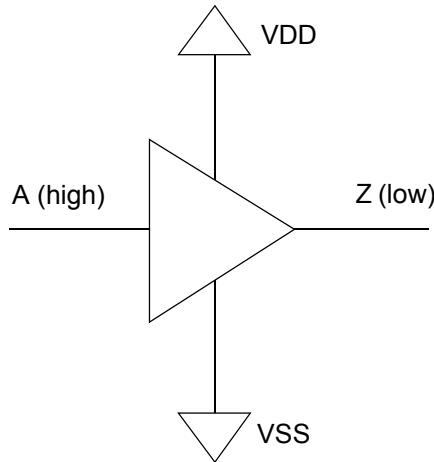
Overdrive Level-Shifter Cell

The cell in [Figure 13-11](#) is known as an overdrive level-shifter cell. To model an overdrive level-shifter cell, specify the `related_ground_pin` attribute and the `input_signal_level` attribute, as shown in the example that follows the figure.

Note:

The area of a multiple-rail level-shifter cell (as shown in [Figure 13-9](#)) is larger than that of an overdrive level-shifter cell (as shown in [Figure 13-11](#)). Keep the area in mind when you design your cell.

Figure 13-11 Overdrive Level-Shifter Cell



```

library(level_shifter_cell_library_example) {
  voltage_map(VDD1, 1.2);
  voltage_map(VDD, 0.8);
  voltage_map(VSS, 0.0);
  operating_conditions(XYZ) {
    process : 1.0;
    voltage : 3.0;
    temperature : 25.0;
  }
}

```

```

}

default_operating_conditions : XYZ;

cell(Buffer_Type_HL_Level_shifter) {
    is_level_shifter : true;
    level_shifter_type : HL ;

    pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
        std_cell_main_rail : true;
    }
    pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
    }
    leakage_power() {
        when : "!A";
        value : 1.5;
        related_pg_pin : VDD;
    }

    pin(A) {
        direction : input;
    /* Defining the input_signal_level attribute identifies an Overdrive
    Level Shifter cell */
        input_signal_level : VDD1;
        related_power_pin : VDD;
        related_ground_pin : VSS;
        input_voltage_range ( 1.1 , 1.3);
    }

    pin(Z) {
        direction : output;
        related_power_pin : VDD;
        related_ground_pin : VSS;
        function : "A";
        power_down_function : "!VDD + VSS";
        output_voltage_range (0.6 , 0.9);

        timing() {
            related_pin : A;
            cell_rise(template) {
                ...
            }
            cell_fall(template) {
                ...
            }
            rise_transition(template) {
                ...
            }
            fall_transition(template) {
                ...
            }
        }
    }
}

```

```

        }
    }
    internal_power() {
        related_pin : A;
        related_pg_pin : VDD;
        ...
    }
} /* end pin group*/
...
} /*end cell group*/
...
} /*end library group */

```

Level-Shifter Cell With Virtual Bias Pins

This section provides an example of a level-shifter cell with virtual bias pins and two n-well regular wells for substrate-bias modeling.

```

library (example_multi_rail_with_bias_pins) {
...
    cell ( level_shifter ) {
        pg_pin ( vdd1 ) {
            pg_type : primary_power ;
            ...
        }
        pg_pin ( vdd2 ) {
            pg_type : primary_power ;
            ...
        }
        pg_pin ( vss ) {
            pg_type : primary_ground ;
            ...
        }
        pg_pin ( vpw ) {
            pg_type : pwell ;
            ...
        }
        pg_pin ( vnwl ) {
            pg_type : nwell ;
            ...
        }
        pg_pin ( vnw2 ) {
            pg_type : nwell ;
            ...
        }
        pin ( I ) {
            direction : input;
            related_power_pin : vdd1
            related_ground_pin : vss
            related_bias_pin : "vnwl vpw"
            ...
        }
    }
}

```

```

pin ( Z ) {
    direction : output;
    function : "I";
    related_power_pin : "vdd2" ;
    related_ground_pin : "vss" ;
    related_bias_pin : "vnw2 vpw";
    power_down_function : "!vdd1 + !vdd2 + vss + !vnw1 + !vnw2 + vpw";
    ...
}
} /* End of cell group */
}/* End of library group */

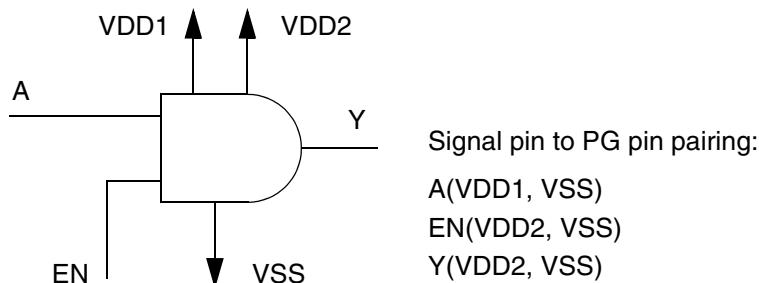
```

Enable Level-Shifter Cell

[Figure 13-12](#) shows the schematic of a basic enable level-shifter cell. The A signal pin is linked to the VDD1 and VSS power and ground pin pair, and the EN signal pin and the Y signal pin are linked to the VDD2 and VSS power and ground pin pair. The enable pin is linked to VDD2.

The example that follows the figure shows a library containing an enable level-shifter cell. To prevent a short between VDD1 and VSS, pin A has the `alive_during_power_up` attribute set to `true` and is active till VDD1 is on.

Figure 13-12 Enable Level-Shifter Cell Schematic



```

library(enable_level_shifter_library_example) {

    voltage_map(VDD1, 0.8);
    voltage_map(VDD2, 1.2);
    voltage_map(VSS, 0.0);

    operating_conditions(XYZ) {
        voltage : 3.0;
        ...
    }
    default_operating_conditions : XYZ;

    cell(Enable_Level_Shifter) {
        is_level_shifter : true;
    }
}

```

```

level_shifter_type : LH ;

pg_pin(VDD1) {
    voltage_name : VDD1;
    pg_type : primary_power;
    std_cell_main_rail : true;
}
pg_pin(VSS) {
    voltage_name : VSS;
    pg_type : primary_ground;
}
pg_pin(VDD2) {
    voltage_name : VDD2;
    pg_type : primary_power;
}

leakage_power() {
    when : "!A";
    value : 1.5;
    related_pg_pin : VDD1;
}
leakage_power() {
    when : "!A";
    value : 1.5;
    related_pg_pin : VDD2;
}

pin(A) {
    direction : input;
    related_power_pin : VDD1;
    related_ground_pin : VSS;
    input_voltage_range ( 0.7 , 0.9 );
    level_shifter_data_pin : true;
}

pin(EN) {
    direction : input;
    related_power_pin : VDD2;
    related_ground_pin : VSS;
    input_voltage_range ( 1.1 , 1.3 );
    level_shifter_enable_pin : true;
}

pin(Y) {
    direction : output;
    related_power_pin : VDD2;
    related_ground_pin : VSS;
    function : "A * EN";
    power_down_function : "!VDD1 + !VDD2 + VSS";
    output_voltage_range ( 1.1 , 1.3 );
    timing() {
        related_pin : "A EN";
    }
}

```

```

    cell_rise(template) {
    ...
}
cell_fall(template) {
...
}
rise_transition(template) {
...
}
fall_transition(template) {
...
}
}

internal_power() {
    related_pin : "A EN";
    related_pg_pin : VDD1;
...
}
internal_power() {
    related_pin : "A EN";
    related_pg_pin : VDD2;
...
}

}/* end pin group*/
...
}/*end cell group*/
...
}/*end library group*/

```

Clamping in Latch-Based Enable Level-Shifter Cell

[Table 13-4](#) shows the truth table for a latch-based enable level-shifter cell with two enable pins, EN1 and EN2. The example that follows the table shows the enable level-shifter cell modeled with the clamp function attributes.

Table 13-4 Truth Table for a Latch-Based Enable Level-Shifter Cell With Two Enable Pins

A	EN1	EN2	Y	Function
1/0	1	0	1/0	Level shifter
-	0	1	1	Clamp to 1
-	0	0	Qn-1	Latch
-	1	1	?	Forbidden

```
cell(clamp_1_latch_ELS) {
```

```
is_level_shifter : true ;
...
pg_pin(VDD1) {
    pg_type : primary_power;
    ...
}
pg_pin(VSS1) {
    pg_type : primary_ground;
    ...
}
pg_pin(VDD2) {
    pg_type : primary_power;
    std_cell_main_rail : true;
    ...
}
pg_pin(VSS2) {
    pg_type : primary_ground;
    ...
}
latch (IQ, IQN) {
    data_in : A;
    enable : EN1;
    preset : EN2;
}
pin (A) {
    related_power_pin : VDD1;
    related_ground_pin : VSS1;
    direction : input;
    level_shifter_data_pin : true ;
    ...
}
pin (EN1) {
    related_power_pin : VDD2;
    related_ground_pin : VSS2;
    direction : input;
    level_shifter_enable_pin : true ;
    ...
}
pin (EN2) {
    related_power_pin : VDD2;
    related_ground_pin : VSS2;
    direction : input;
    level_shifter_enable_pin : true ;
    ...
}
pin (Y) {
    related_power_pin : VDD2;
    related_ground_pin : VSS2;
    direction : output;
    clamp_1_function: "!EN1 * EN2";
    clamp_latch_function: "!EN1 * !EN2";
    illegal_clamp_function: "EN1 * EN2";
    function : "IQ";
```

```

    ...
}
...
}
```

Level-Shifter Cell Not Powered by Switching Power Domains

The following example is a typical library description of a buffer-type, enable-type, and an overdrive level-shifter cell that also include a dummy `std_cell_main_rail` PG pin.

```

library(mylib) {
    delay_model : table_lookup;
    /* unit attributes */
    time_unit : "1ns";
    voltage_unit : "1V";
    current_unit : "1mA";
    pulling_resistance_unit : "1kohm";
    leakage_power_unit : "1nW";
    capacitive_load_unit (1.0,pf);
    voltage_map(VDD1, 0.80); /* secondary power */
    voltage_map(VDD2, 1.20); /* secondary power */
    voltage_map(VDD, 0.80); /* primary power */
    voltage_map(VSS, 0.0); /* primary ground */
    nom_process : 1.0;
    nom_voltage : 0.8;
    nom_temperature : 125;
    /* operation conditions */
    operating_conditions(XYZ) {
        process      : 1;
        temperature : 125;
        voltage     : 0.8;
        tree_type   : balanced_tree
    }
    default_operating_conditions : XYZ;
    /* threshold definitions */
    slew_lower_threshold_pct_fall : 30.0;
    slew_upper_threshold_pct_fall : 70.0;
    slew_lower_threshold_pct_rise : 30.0;
    slew_upper_threshold_pct_rise : 70.0;
    input_threshold_pct_fall     : 50.0;
    input_threshold_pct_rise     : 50.0;
    output_threshold_pct_fall    : 50.0;
    output_threshold_pct_rise    : 50.0;
    /* default attributes */
    default_cell_leakage_power   : 0.0;
    default_fanout_load         : 1.0;
    default_output_pin_cap       : 0.0;
    default_inout_pin_cap        : 0.1;
    default_input_pin_cap        : 0.1;
    default_max_transition       : 1.0;
    cell(Buffer_LH_Level_shifter) {
        area : 1.0;
```

```

is_level_shifter : true;
level_shifter_type : LH ;
pg_pin(VDD1) {
    voltage_name : VDD1;
    pg_type : primary_power;
}
pg_pin(VDD2) {
    voltage_name : VDD2;
    pg_type : primary_power;
}
pg_pin(VDD) {
    voltage_name : VDD;
    pg_type : primary_power;
    std_cell_main_rail : true;
}
pg_pin(VSS) {
    voltage_name : VSS;
    pg_type : primary_ground;
}
pin(A) {
    direction : input;
    capacitance : 1.0;
    related_power_pin : VDD1;
    related_ground_pin : VSS;
    input_voltage_range ( 0.7 , 0.9);
    internal_power() {
        rise_power (scalar) { values ("1.0");}
        fall_power (scalar) { values ("1.0");}
    }
}
pin(Z) {
    direction : output;
    related_power_pin : VDD2;
    related_ground_pin : VSS;
    function : "A";
    power_down_function : "!VDD1 + !VDD2 + VSS";
    output_voltage_range (1.1 , 1.3);
    timing() {
        related_pin : A;
        cell_rise(scalar) { values ("0.1");}
        rise_transition (scalar) { values ("0.1");}
        cell_fall(scalar) { values ("0.1");}
        fall_transition (scalar) { values ("0.1");}
    }
} /* end pin group */
} /* end cell group */
cell(Enable_LH_Level_Shifter) {
    area : 1.0;
    is_level_shifter : true;
    level_shifter_type : LH ;
    pg_pin(VDD1) {
        voltage_name : VDD1;
        pg_type : primary_power;
    }
}

```

```

    }
    pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
    }
    pg_pin(VDD2) {
        voltage_name : VDD2;
        pg_type : primary_power;
    }
    pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
        std_cell_main_rail : true;
    }
    pin(A) {
        direction : input;
        related_power_pin : VDD1;
        related_ground_pin : VSS;
        input_voltage_range ( 0.7 , 0.9 );
        level_shifter_data_pin : true;
        capacitance : 1.0;
        internal_power() {
            rise_power(scalar) { values ("1.0"); }
            fall_power(scalar) { values ("1.0"); }
        }
    }
    pin(EN) {
        direction : input;
        related_power_pin : VDD2;
        related_ground_pin : VSS;
        input_voltage_range ( 1.1 , 1.3 );
        level_shifter_enable_pin : true;
        capacitance : 1.0;
        internal_power() {
            rise_power(scalar) { values ("1.0"); }
            fall_power(scalar) { values ("1.0"); }
        }
    }
    pin(Y) {
        direction : output;
        related_power_pin : VDD2;
        related_ground_pin : VSS;
        function : "A * EN";
        power_down_function : "!VDD1 + !VDD2 + VSS";
        output_voltage_range ( 1.1 , 1.3 );
        timing() {
            related_pin : "A EN";
            cell_rise(scalar) { values ("0.1"); }
            rise_transition (scalar) { values ("0.1"); }
            cell_fall(scalar) { values ("0.1"); }
            fall_transition (scalar) { values ("0.1"); }
        }
    }
} /* end pin group */

```

```
    } /* end cell group */
cell(Over_Drive_Level_Shifter) {
    area : 1.0;
    is_level_shifter : true;
    level_shifter_type : HL ;
    input_voltage_range ( 1.1 , 1.3);
    output_voltage_range (0.7 , 0.9);
    pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
        std_cell_main_rail : true;
    }
    pg_pin(VDD1) {
        voltage_name : VDD1;
        pg_type : primary_power;
    }
    pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
    }
    pin(A) {
        direction : input;
        input_signal_level : VDD2; /* Specifying the
input_signal_level attribute identifies an over-drive cell
*/
        related_power_pin : VDD1;
        related_ground_pin : VSS;
        capacitance : 1.0;
    }
    pin(Z) {
        direction : output;
        function : A;
        related_power_pin : VDD1;
        related_ground_pin : VSS;
        power_down_function : "!VDD1 + VSS";
        timing() {
            related_pin : "A"
            cell_rise( scalar ) { values("0.1"); }
            rise_transition( scalar ) { values("0.1"); }
            cell_fall( scalar ) { values("0.1"); }
            fall_transition( scalar ) { values("0.1"); }
        }
    } /* end pin group */
} /* end cell group */
} /* end library group */
```

Level-Shifter Library Checks

The Library Compiler tool performs checks for level-shifter cells and issues an error or warning message if certain conditions occur. For a detailed list of library checks for level-shifter cells, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Isolation Cell Modeling

In partition-based designs with multiple power supplies and voltages, the outputs from a shut-down partition to an active partition must be maintained at predictable signal levels. An isolation cell isolates the shut-down partition from the active partition. The isolation logic ensures that all the inputs to the active partition are clamped to a fixed value.

Note:

All the pin-level attributes to model isolation cells including the clock-isolation and output clamp function (see “[Clamping Isolation Cell Output Pins](#)” on page 13-71) attributes also apply to low-power complex macro cell models. For more information, see “[Low Power Macro Cell Modeling](#)” on page 13-144.

Example 13-6 shows the syntax for modeling the isolation cell, and its pins.

Example 13-6 Isolation Cell Modeling Syntax

```
cell(isolation_cell) {
    is_isolation_cell : true ;
    ...
    pg_pin(pg_pin_name_P) {
        pg_type : primary_power ;
        permit_power_down : true ;
        ...
    }
    pg_pin(pg_pin_name_G) {
        pg_type : primary_ground;
        ...
    }

    pin (data) {
        direction : input;
        isolation_cell_data_pin : true ;
        ...
    }

    pin (enable) {
        direction : input;
        isolation_cell_enable_pin : true ;
        alive_during_partial_power_down : true ;
        ...
    }
}
```

```
...
    pin (output) {
        direction : output;
        alive_during_partial_power_down : true ;
        function : "Boolean Expression";
        power_down_function : "Boolean Expression";
    }
}/* End pin group */

}/* End Cell group */
```

Cell-Level Attribute

This section describes a cell-level attribute of the isolation cells.

is_isolation_cell Attribute

The `is_isolation_cell` attribute identifies a cell as an isolation cell. The valid values of this attribute are `true` or `false`. If not specified, the default is `false`, meaning that the cell is an ordinary standard cell.

Pin-Level Attributes

This section describes the pin-level attributes for the isolation cells.

isolation_cell_data_pin Attribute

The `isolation_cell_data_pin` attribute identifies the data pin of an isolation cell. The valid values are `true` or `false`. If not specified, all the input pins of the isolation cell are considered to be data pins.

isolation_cell_enable_pin Attribute

The `isolation_cell_enable_pin` attribute identifies an enable or control pin of an isolation cell including a clock isolation cell. The valid values are `true` or `false`. The default is `false`.

power_down_function Attribute

The `power_down_function` attribute identifies the Boolean condition to switch off the output pin of the cell (when the cell is inactive due to the external power-pin states). If the `power_down_function` attribute is set to 1, then X is assumed on the pin.

permit_power_down Attribute

The `permit_power_down` attribute indicates that the power pin can be powered down while in the isolation mode. The valid values are `true` or `false`. The default is `true`.

You can also use this attribute to model low-power complex macro cells. For more information, see “[Low Power Macro Cell Modeling](#)” on page 13-144.

alive_during_partial_power_down Attribute

The `alive_during_partial_power_down` attribute indicates that the pin with this attribute is active while the isolation cell is partially powered down, and the corresponding power and ground rails are not considered as the power reference. The valid values are `true` and `false`. The default is `true`, and in the default setting, the UPF isolation supply set is the power reference.

You can also use this attribute to model low-power complex macro cells. For more information, see “[Low Power Macro Cell Modeling](#)” on page 13-144.

alive_during_power_up Attribute

The optional `alive_during_power_up` attribute specifies an active data pin that is powered by a more always-on supply. The default is `false`. If set to `true`, it indicates that the data pin is active while its related power pin is active.

You can specify this attribute only in a `pin` group where the `isolation_cell_data_pin` or the `level_shifter_data_pin` attribute is set to `true`.

Attribute Dependencies

The isolation cell attributes have the following dependencies:

- When the control pin of an isolation cell is activated, the output becomes a constant.
- The control pin of an isolation cell, that permits partial power down must be included in the Boolean expression of the `power_down_function` attribute for the same output. The control pin blocks the terms that use the powered-down rail, to set to `true`. To generate the output in the isolation mode, the active power rail is used.

Therefore, an isolation cell cannot be partially powered down if the `power_down_function` expression of its power pin does not include the `isolation_cell_enable_pin` attribute set.

Note:

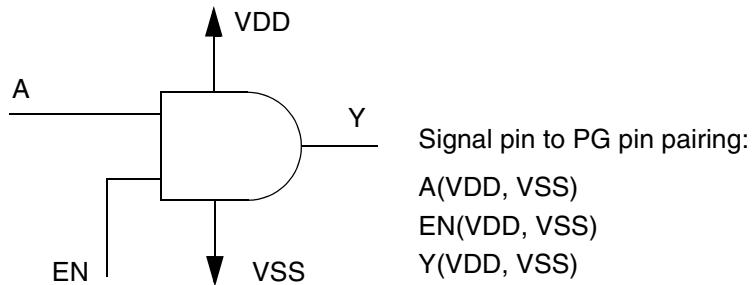
The Library Compiler tool allows only one pin with the `isolation_cell_enable_pin` attribute in accordance to IEEE Standard 1801 but allows other nonisolation type enable pins to coexist in the cell.

Isolation Cell Examples

Unlike level-shifter cells, isolation cells cannot shift voltage levels. All other characteristics are the same between a level-shifter cell and an isolation cell.

[Figure 13-13](#) shows a schematic and a library description of an isolation cell. The library describes only the portion related to the power and ground pin syntax. In the figure, the A, EN, and Y signal pins are associated to the VDD and VSS power and ground pin pair. The example shows a library with an isolation cell.

Figure 13-13 Isolation Cell Schematic



```
library(isolation_cell_library_example) {
    voltage_map(VDD, 1.0);
    voltage_map(VSS, 0.0);

    operating_conditions(XYZ) {
        voltage : 1.0;
        ...
    }
    default_operating_conditions : XYZ;

    cell(Isolation_Cell) {
        is_isolation_cell : true;
        dont_touch : true;
        dont_use : true;

        pg_pin(VDD) {
            voltage_name : VDD;
            pg_type : primary_power;
        }

        pg_pin(VSS) {
            voltage_name : VSS;
            pg_type : primary_ground;
        }
    }
}
```

```

leakage_power() {
    when : "!A";
    value : 1.5;
    related_pg_pin : VDD;
}

pin(A) {
    direction : input;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    isolation_cell_data_pin : true;
}

pin(EN) {
    direction : input;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    isolation_cell_enable_pin : true;
}

pin(Y) {
    direction : output;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    function : "A * EN";
    power_down_function : "!VDD + VSS";
    timing() {
        related_pin : "A EN";
        cell_rise(template) {
            ...
        }
        cell_fall(template) {
            ...
        }
        rise_transition(template) {
            ...
        }
        fall_transition(template) {
            ...
        }
    }
    internal_power() {
        related_pin : A;
        related_pg_pin : VDD;
        ...
    }
}

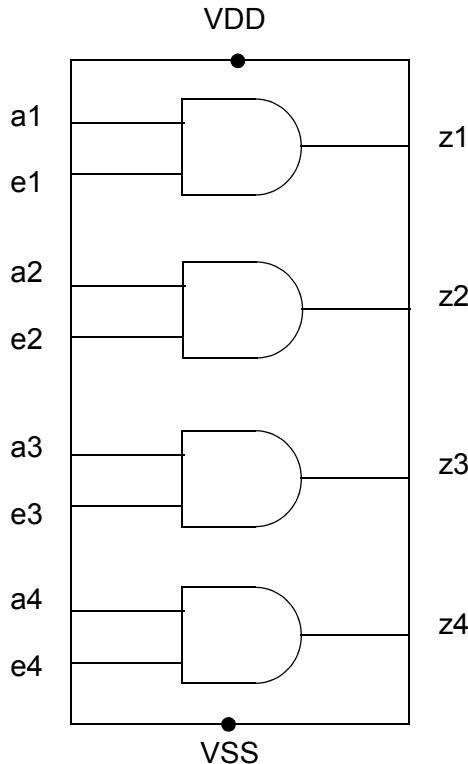
/* end pin group*/
...
}/*end cell group*/
...
}/*end library group*/

```

Multibit Isolation Cell

[Figure 13-14](#) shows a multibit isolation cell with four data input pins, four enable pins for each of the data input pins, and four data output pins. Each of the data outputs is a linear function of the respective data input. The example following the figure is a model of the isolation cell.

Figure 13-14 4-bit Isolation Cell



```
cell (4_bit_iso) {
    is_isolation_cell : true;
    pg_pin (VDD) {
        pg_type : primary_power;
        voltage_name : VDD;
    }
    pg_pin (VSS) {
        pg_type : primary_ground;
        voltage_name : VSS;
    }
    ...
    pin (a1) {
        isolation_cell_data_pin : true;
        related_power_pin : VDD;
        related_ground_pin : VSS;
    }
    ...
}
```

```
        }
    pin (a2) {
        isolation_cell_data_pin : true;
        related_power_pin : VDD;
        related_ground_pin : VSS;
        ...
    }
    pin (a3) {
        isolation_cell_data_pin : true;
        related_power_pin : VDD;
        related_ground_pin : VSS;
        ...
    }
    pin (a4) {
        isolation_cell_data_pin : true;
        related_power_pin : VDD;
        related_ground_pin : VSS;
        ...
    }

    pin (e1) {
        isolation_cell_enable_pin : true;
        related_power_pin : VDD;
        related_ground_pin : VSS;
        ...
    }
    pin (e2) {
        isolation_cell_enable_pin : true;
        related_power_pin : VDD;
        related_ground_pin : VSS;
        ...
    }
    pin (e3) {
        isolation_cell_enable_pin : true;
        related_power_pin : VDD;
        related_ground_pin : VSS;
        ...
    }
    pin (e4) {
        isolation_cell_enable_pin : true;
        related_power_pin : VDD;
        related_ground_pin : VSS;
        ...
    }

    pin (z1) {
        related_power_pin : VDD;
        related_ground_pin : VSS;
        power_down_function : !VDD + VSS;
        function : a1 * e1;
        clam_0_function : !e1;
        ...
    }
}
```

```

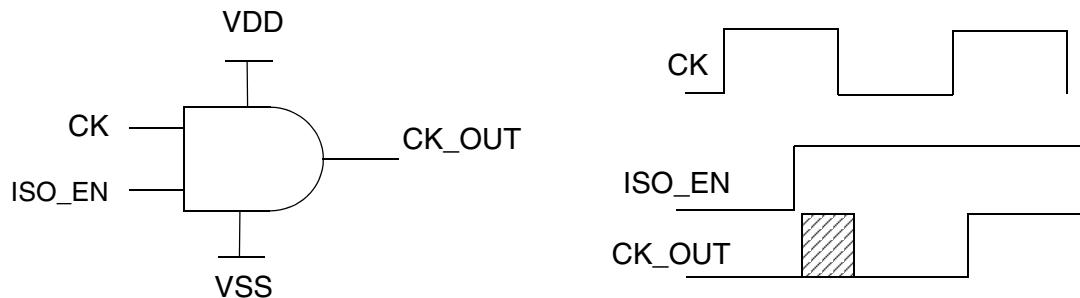
pin (z2) {
    related_power_pin : VDD;
    related_ground_pin : VSS;
    power_down_function : !VDD + VSS;
    function : a2 * e2;
    clam_0_function : !e2;
...
}
pin (z3) {
    related_power_pin : VDD;
    related_ground_pin : VSS;
    power_down_function : !VDD + VSS;
    function : a3 * e3;
    clam_0_function : !e3;
...
}
pin (z4) {
    related_power_pin : VDD;
    related_ground_pin : VSS;
    power_down_function : !VDD + VSS;
    function : a4 * e4;
    clam_0_function : !e4;
...
}
}

```

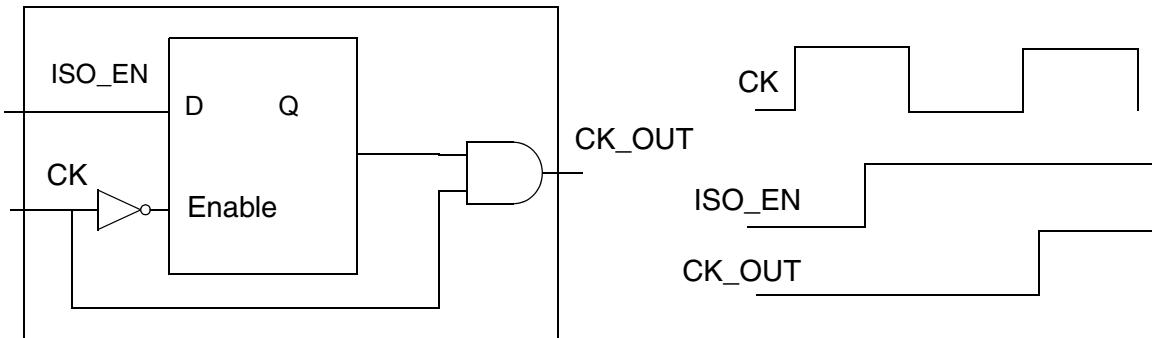
Clock-Isolation Cell Modeling

Using combinational isolation cells to isolate clock signals might result in phase-clipped outputs. [Figure 13-15](#) shows an AND isolation cell. The phase of the output clock signal, CK_OUT, becomes clipped, depending on the arrival time of the isolation-enable signal, ISO_EN, with respect to the input clock signal, CK.

Figure 13-15 Clock Isolation With AND Isolation Cell



To avoid phase-clipping, use clock-isolation cells for clock isolation. [Figure 13-16](#) shows a schematic and a typical output of a clock-isolation cell.

Figure 13-16 Clock Isolation Cell Schematic

Note:

The phase of the output clock signal, CK_OUT, is not clipped.

The modeling syntax of the clock-isolation cell and its pins is as follows:

```
cell (cell_name) {
    is_clock_isolation_cell : true;
    pin(input_pin) {
        clock_isolation_cell_clock_pin : true;
        direction : input;
        ...
    }
    pin(input_pin) {
        isolation_cell_enable_pin : true;
        direction : input;
        ...
    }
    pin(output_pin_name) {
        direction : output;
        ...
    }
    ...
}
```

Cell-Level Attribute

This section describes the cell-level attribute of clock-isolation cells.

is_clock_isolation_cell Attribute

The `is_clock_isolation_cell` attribute identifies a cell as a clock-isolation cell. The default is `false`, meaning that the cell is a standard cell.

Pin-Level Attributes

This section describes the pin-level attributes for clock-isolation cells.

isolation_cell_enable_pin Attribute

The `isolation_cell_enable_pin` attribute identifies an enable or control pin of a clock-isolation cell. The default is `false`.

clock_isolation_cell_clock_pin Attribute

The `clock_isolation_cell_clock_pin` attribute identifies an input clock pin of a clock-isolation cell. The default is `false`.

Clock Isolation Cell Examples

[Example 13-7](#) shows a library description of the clock-isolation cell in [Figure 13-16](#) on [page 13-68](#).

Example 13-7 Library Description of the Clock-Isolation Cell

```
library (my_lib) {
    ...
    cell(ck_iso_cell) {
        is_clock_isolation_cell : true;
        ...
        pg_pin (VDD) {
            pg_type : primary_power;
            voltage_name : VDD;
        }
        pg_pin (VSS) {
            pg_type : primary_ground;
            voltage_name : VSS;
        }
        statetable(" CK ISO_EN ", "Q ") {
            table : " L L : - : L , \
                      L H : - : H , \
                      H - : - : N ";
        }
        pin(CK) {
            clock_isolation_cell_clock_pin : true;
            direction : input;
            related_ground_pin : VSS;
            related_power_pin : VDD;
            ...
        }
        pin(ISO_EN) {
            isolation_cell_enable_pin : true;
            direction : input;
            related_ground_pin : VSS;
            related_power_pin : VDD;
            ...
        }
    }
}
```

```

pin(CK_OUT) {
    direction : output;
    power_down_function : "!VDD + VSS";
    related_ground_pin : VSS;
    related_power_pin : VDD;
    state_function : "CK * Q";
    timing() {
        related_pin : "CK";
        timing_sense : "positive_unate";
        ...
    }
    ...
}
}

```

[Example 13-8](#) shows the clock-isolation cell also modeled as a clock-gating cell. To model a cell as both a clock-isolation cell and a clock-gating cell, use both the clock-isolation and clock-gating attributes.

Example 13-8 Example of a Cell Modeled as Both Clock-Isolation and Clock-Gating Cell

```

cell(ICG_CK_ISO_CELL) {
    is_clock_isolation_cell : true;
    clock_gating_integrated_cell : "latch_posedge";
    ...
    pg_pin (VDD) {
        pg_type : primary_power;
        voltage_name : VDD;
    }
    pg_pin (VSS) {
        pg_type : primary_ground;
        voltage_name : VSS;
    }
    statetable(" CK EN ", "Q ") {
        table : " L L : - : L , \
                  L H : - : H , \
                  H - : - : N ";
    }
    pin(CK) {
        clock_isolation_cell_clock_pin : true;
        clock_gate_clock_pin : true;
        direction : input;
        related_ground_pin : VSS;
        related_power_pin : VDD;
        ...
    }
}

```

```

pin(EN) {
    isolation_cell_enable_pin : true;
    clock_gate_enable_pin : true;
    direction : input;
    related_ground_pin : VSS;
    related_power_pin : VDD;
    ...
}
pin(CK_OUT) {
    direction : output;
    clock_gate_output_pin : true;
    power_down_function : "!VDD + VSS";
    related_ground_pin : VSS;
    related_power_pin : VDD;
    state_function : "CK * Q";
    timing() {
        related_pin : "CK";
        timing_sense : "positive_unate";
        ...
    }
    ...
}

```

Clamping Isolation Cell Output Pins

In isolation cells with multiple enable pins, clamping the outputs might be required to maintain them at particular logic levels, such as 0,1, z, or a previously-latched value.

The clamp function modeling syntax for the isolation cell and its pins is as follows:

```

cell (cell_name) {
    is_isolation_cell : true;
    ...
    pin(input_pin) {
        direction : input;
        isolation_cell_data_pin : true;
        ...
    }
    pin(input_pin) {
        direction : input;
        isolation_cell_enable_pin : true;
        ...
    }
    pin(input_pin) {
        direction : input;
        isolation_cell_enable_pin : true;
        ...
    }
}

```

```

pin(output_pin_name) {
    direction : output;
    clamp_0_function : "Boolean expression" ;
    clamp_1_function : "Boolean expression" ;
    clamp_z_function : "Boolean expression" ;
    clamp_latch_function : "Boolean expression" ;
    illegal_clamp_condition : "Boolean expression" ;
    ...
}
...
}

```

Pin-Level Attributes

This section describes the pin-level clamp function attributes to clamp the isolation cell output pins.

clamp_0_function Attribute

The `clamp_0_function` attribute specifies the input condition for the enable pins of an isolation cell when the output clamps to 0.

clamp_1_function Attribute

The `clamp_1_function` attribute specifies the input condition for the enable pins of an isolation cell when the output clamps to 1.

clamp_z_function Attribute

The `clamp_z_function` attribute specifies the input condition for the enable pins of an isolation cell when the output clamps to z.

clamp_latch_function Attribute

The `clamp_latch_function` attribute specifies the input condition for the enable pins of an isolation cell when the output clamps to the previously latched value.

illegal_clamp_condition Attribute

The `illegal_clamp_condition` attribute specifies the invalid condition for the enable pins of an isolation cell. If the `illegal_clamp_condition` attribute is not specified, the invalid condition does not exist.

Note:

All the input pins used in the Boolean expressions of the `clamp_0_function`, `clamp_1_function`, `clamp_z_function`, and `clamp_latch_function` attributes must be enable pins with the `isolation_cell_enable_pin` attribute set to true. The Boolean expressions of the `clamp_0_function`, `clamp_1_function`,

`clamp_z_function`, `clamp_latch_function`, and `illegal_clamp_condition` attributes must be mutually exclusive.

Example of Clamping in Isolation Cell

[Table 13-4](#) shows the truth table for an isolation cell with two enable pins, EN1 and EN2. [Example 13-9](#) shows the isolation cell modeled with a clamp function attribute.

Table 13-5 Truth Table for an Isolation Cell With Two Enable Pins

A	EN1	EN2	Y	Function
0/1	1	1	0/1	AND
-	0	1	0	Clamp to 0
-	1	0	0	Clamp to 0
-	0	0	0	Clamp to 0

Example 13-9 Using a Clamp Function Attribute to Model an Isolation Cell

```
cell(ISO_clamp_0) {
    is_isolation_cell : true;
    ...
    pin(A) {
        direction : input;
        capacitance : 1.0;
        isolation_cell_data_pin : true;
        ...
    }
    pin(EN1) {
        direction : input;
        capacitance : 1.0;
        isolation_cell_enable_pin : true;
        ...
    }
    pin(EN2) {

        direction : input;
        capacitance : 1.0;
        isolation_cell_enable_pin : true;
        ...
    }
    pin(Y) {
        direction : output;
        function : "A*EN1*EN2";
        clamp_0_function : "! (EN1*EN2)";
        ...
    }
}
```

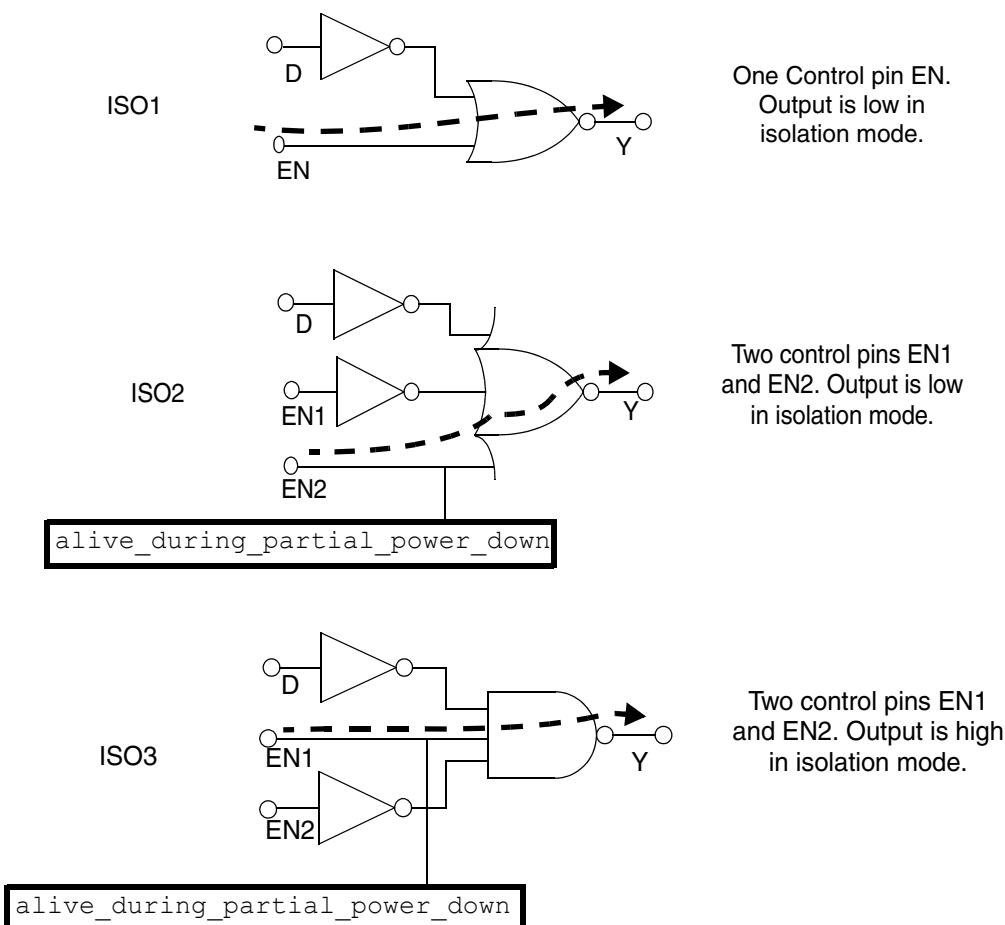
{}

Isolation Cells With Multiple Control Pins

In partition-based designs, output pins of an isolation-cell need to be maintained at predefined levels for a significant period of time. In such designs, using isolation cells with multiple control or enable pins reduces the leakage current. Multiple control pins allow an isolation cell to be partially powered down. For example, in an isolation cell with two control pins, one control pin controls the output level. The other control pin is used to partially power-down the isolation cell while the output level is maintained.

Figure 13-17 shows the gate-level diagrams of isolation cells with multiple control pins. In each of the three diagrams, the dotted-line arrow traces the path from the isolation-control pin to the output pin.

Figure 13-17 Gate-Level Diagrams of Isolation Cells With Multiple Control Pins



The isolation control pins permit the power pins (not shown in [Figure 13-17](#)) to power down. The `alive_during_partial_power_down` attribute on the isolation-control pins indicate that these pins remain active even when the corresponding power pins are powered down. [Example 13-10](#) shows the partially powered down model of the cell, ISO1, depicted in [Figure 13-17](#). [Table 13-6](#) shows the Boolean expressions for the `function` and `power_down_function` attributes of the isolation cells depicted in [Figure 13-17](#).

Example 13-10 Partially Powered Down Model of the ISO1 Cell

```
cell ( ISO1 ) {
    is_isolation_cell : true;
    pg_pin ( VDD ) {
        pg_type : primary_power;
        permit_power_down : true
    }
    pg_pin ( VSS ) {
        pg_type : primary_ground ;
    }
    pin ( D ) {
        isolation_cell_data_pin : true ;
    }
    pin ( EN) {
        direction : input
        isolation_cell_enable_pin : true ;
        alive_during_partial_power_down : true ;
    }
    pin ( Y ) {
        direction : output ;
        alive_during_partial_power_down : true ;
        function : D * !EN ;
        power_down_function : (!VDD * !EN) + VSS ;
    }
}
```

Note:

By default, the `related_power_pin` and `related_ground_pin` attributes are mapped to VDD and VSS, respectively. The `related_power_pin` and `related_ground_pin` attributes are not specified for the input and output pins in [Example 13-10](#).

Table 13-6 Boolean Expressions for the function and power_down_function Attributes of the Isolation Cells in [Figure 13-17](#)

Cell	function (Output Y)	power_down_function (Output Y)
ISO1	D * !EN	(!VDD * !EN) + VSS
ISO2	D * EN1 * !EN2	(!VDD * !EN2) + VSS
ISO3	D + !EN1 + EN2	!VDD + (VSS + EN1)

Pins with the `alive_during_partial_power_down` attribute do not require the power pins to be active. For example, for the cell ISO2, the isolation-control pin, EN2, does not require VDD to be active. This pin controls the power-down of VDD through the Boolean expression of the `power_down_function` attribute as shown in [Table 13-6](#). The other control pin, EN1, without the `alive_during_partial_power_down` attribute requires both VDD and VSS to be active, and is not included in the Boolean expression of the `power_down_function` attribute.

In each of the isolation cells in [Figure 13-17](#), the `permit_power_down` attribute is set on a power pin when there is at least one pin with the `isolation_cell_enable_pin` attribute. For example, for the cell ISO2, the `permit_power_pin` attribute is set on the power pin, VDD, given the `isolation_cell_enable_pin` attribute is set on the pin, EN2.

Isolation Cell Library Checks

The Library Compiler tool checks for isolation cells including clock isolation cells and isolation cells with multiple control pins and issues an error or warning message if certain conditions occur. For a detailed list of library checks for isolation cells, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter of the *Library Quality Assurance System User Guide*.

Switch Cell Modeling

Switch cells, also known as multithreshold complementary metal oxide semiconductor cells (power-switch cells), are used to reduce power. They are divided into the following two classes:

- Coarse grain

There are two types of coarse-grain switch cells, header switch cells, and footer switch cells. The header switch cells control the power nets based on a PMOS transistor, and the footer switch cells control the ground nets based on an NMOS transistor. The coarse grain cell is a switch that drives the power to other logic cells. It is used as a big switch to the supply rails and to turn off design partitions when the relative logic is inactive. Therefore, coarse-grain switch cells can reduce the leakage of the inactive logic.

In addition, coarse-grain switch cells can also have the properties of a fine-grain switch cell. For example, they can act as a switch and have output pins that might or might not be shut off by the internal switch pins.

- Fine grain

To reduce the leakage power, the fine-grain switch cell includes an embedded switch pin that is used to turn off the cell when it is inactive.

The Library Compiler syntax supports only fine-grain switch cells for macro cells.

Coarse-Grain Switch Cells

Coarse-grain switch cells must have the following properties:

- They must be able to model the condition under which the cell turns off the controlled design partition or the cells connected in the output power pin's fanout logical cone. This is modeled with a switching function based on special switch signal pins as well as a separate but related power-down function based on power pin inputs.
- They must be able to model the “acknowledge” output pins (output pins whose signal is used to propagate the switch signal to the next-switch cell or to a power controller input's logic cloud). Timing is also propagated from the input switch pin to the acknowledge output pins.
- They must have at least one virtual output power and ground pin (virtual VDD or virtual VSS), one regular input power and ground pin (VDD or VSS), and one switch input pin. There is no limit on the number of pins a coarse-grain cell can have.

The following describes a simple coarse-grain switch header cell and a simple coarse-grain switch footer cell:

- Header cell: one switch input pin, one VDD (power) input power and ground pin, and one virtual VDD (internal power) output power and ground pin
- Footer cell: one switch input pin, one virtual VSS (internal ground) output power and ground pin, and one VSS (ground) input power and ground pin
- They can have multiple switch pins and multiple acknowledge output pins.
- They must have the steady state current (I/V) information to determine the resistance value when the switch is on.
- The timing information can be specified for coarse-grain switch cells on the output pins, and it can be state dependent for switch pins.

The power output pins in a coarse-grain switch cell can have the following two states:

- Awake or On

In this state, the input power level is transmitted through the cell to either a 1 or 0 on the output power pin, depending on other prior switch cells in series settings.

- Off

In this state, the sleep pin deactivates the pass-through function, and the output power pin is set to X.

Coarse-Grain Switch Cell Syntax

The following syntax is a portion of the coarse-grain switch cell syntax:

```

library(coarse_grain_library_name) {
...
lu_table _template (template_name)
    variable_1 : input_voltage;
    variable_2 : output_voltage;
    index_1 ( float, ... );
    index_2 ( float, ... );
}
...
cell(cell_name) {
    switch_cell_type : coarse_grain;
    ...
    pg_pin (VDD/VSS pin name) {
        pg_type : primary_power | primary_ground;
        direction : input ;
        ...
    }
/* Virtual power and ground pins use "switch_function" to describe the
logic to shut off the attached design partition */

    pg_pin (virtual VDD/VSS pin name) {
        pg_type : internal_power | internal_ground;
        direction: output;
        ...
        switch_function : "function_string";
        pg_function : "function_string";
    }
    dc_current (dc_current_name) {
        related_switch_pin : input_pin_name;
        related_pg_pin : VDD pin name;
        related_internal_pg_pin : Virtual VDD;

        values("float, ...");
    }
    pin (input_pin_name) {
        direction : input;
        switch_pin : true;
        ...
    }
    ...
/* The acknowledge output pin uses "function" to represent the propagated
switching
signal
*/
    pin(acknowledge_output_pin_name) {
        ...
        function : "function_string";
        power_down_function : "function_string";
        direction : output;
        ...
    }
}

```

```
    } /* end pin group */
} /* end cell group */
```

You can use the following syntax for intrinsic parasitic models.

```
switch_cell_type : coarse_grain;
dc_current (template_name) {
    related_switch_pin : pin_name;
    related_pg_pin : pg_pin_name;
    related_internal_pg_pin : pg_pin_name;
    index_1 ( "float, ..." );
    index_2 ( "float, ..." );
    values ( "float, ..." );
}
```

Library-Level Group

The following attribute is a library-level attribute for switch cells.

lu_table_template Group

The library-level `lu_table_template` group models the templates for the steady state current information that is used within the `dc_current` group. The `input_voltage` value specifies input voltage values for the switch pin. The `output_voltage` value specifies the output voltage values of the switch pin. The `input_voltage` and `output_voltage` values are the absolute gate voltage and absolute drain voltage, respectively, when a CMOS transistor is used to model a power-switch cell.

The `dc_current` group, which is used for steady state current modeling, must be defined at the cell level. It is defined by two indexes: `index_1` and `index_2`. The `index_1` attribute represents a string that includes a comma-separated set of N values, where N represents the table rows. The values define the voltage levels measured at either the input voltage or the output voltage. When referring to the input voltage, the voltage level is measured at the related switch pin, and the set of N values must have at least two values for N. When referring to the output voltage, the voltage level is measured at the related internal PG pin, and the set of N values must have at least three values for N. This voltage level is related to a common reference ground for the cell. The set of N `index_1` values must increase monotonically or the Library Compiler tool issues an error.

The `index_2` attribute represents a string that includes a comma-separated set of M values, where M represents the table columns. The values define the voltage levels that are specified at either the input voltage or the output voltage. When referring to the input voltage, the voltage level is measured at the related switch pin, and the set of M values must have at least two values for M. When referring to the output voltage, the voltage level is measured at the related internal PG pin, and the set of M values must have at least three values for M. This voltage level is related to a common reference ground for the cell. The set of M `index_2` values must increase monotonically or the Library Compiler tool issues an error.

Cell-Level Attribute and Group

The following cell-level attribute and group apply to coarse-grain switch cells.

switch_cell_type Attribute

The `switch_cell_type` attribute specifies the switch cell type so that it is not indirectly derived from the cell modeling description. Valid values are `coarse_grain` and `fine_grain`.

dc_current Group

The cell-level `dc_current` group models the steady state current information, similar to the `lu_table_template` group. The table is used to specify the DC current through the cell's output pin (generally the `related_internal_pg_pin`) in the current units specified at the library level using the `current_unit` attribute.

The `dc_current` group includes the `related_switch_pin`, `related_pg_pin`, and `related_internal_pg_pin` attributes, which are described in the following sections.

related_switch_pin Attribute

The `related_switch_pin` string attribute specifies the name of the related switch pin for the coarse-grain switch cell.

related_pg_pin Attribute

The `related_pg_pin` string attribute is used to specify the name of the power and ground pin that represents the VDD or VSS power source.

related_internal_pg_pin Attribute

The `related_internal_pg_pin` string attribute is used to specify the name of the power and ground pin that represents the virtual VDD or virtual VSS power source.

Pin-Level Attributes

The following attributes are pin-level attributes for coarse-grain switch cells.

switch_function Attribute

The `switch_function` string attribute identifies the condition when the attached design partition is turned off by the input `switch_pin`.

For a coarse-grain switch cell, the `switch_function` attribute can be defined at both controlled power and ground pins (virtual VDD and virtual VSS for `pg_pin`) and the output pins. It identifies the signal pins that can turn the power pin on.

When the `switch_function` attribute is defined in the controlled power and ground pin, it is used to specify the Boolean condition under which the cell switches off (or drives an X to) the controlled design partitions, including the traditional signal input pins only with no related power pins to this output.

switch_pin Attribute

The `switch_pin` attribute is a pin-level Boolean attribute. When it is set to true, it is used to identify the pin as the switch pin of a coarse-grain switch cell.

function Attribute

The `function` attribute in a pin group defines the value of an output pin or inout pin in terms of the input pins or inout pins in the cell group or model group. The `function` attribute describes the Boolean function of only nonsleep input signal pins.

pg_function Attribute

The `pg_function` attribute specifies the Boolean function of a virtual PG pin as a function of the cell's actual input power and ground pins. Specify the `pg_function` attribute in the `pg_pin` group.

Note:

Use the `pg_function` attribute to model:

- The internal and virtual power output pins that propagate power to the coarse-grain and fine-grained switches (as a function of the actual input power and ground pins).

For more information about modeling fine-grained switch cells, see “[Fine-Grained Switch Support for Macro Cells](#)” on page 13-82.

- The PG pins of internal voltage converters of macro cells.

Do not use the `switch_cell_type` and `switch_function` attributes to model macro cells with internal voltage converters.

For more information about modeling macro cells with internal PG pins, see “[Modeling Macro Cells With Internal PG Pins](#)” on page 13-141.

power_down_function Attribute

The `power_down_function` string attribute is used to identify the condition under which the cell's signal output pin is switched off (when the cell is in off mode due to the external power pin states). If `power_down_function` is set to 1, then X is assumed on the pin.

Note:

If certain signal pins are not associated to a power and ground pin (using the `related_power_pin` and the `related_ground_pin` attributes), the Library Compiler

tool associates those signal pins to the primary power rails defined on the cell and issues the following warning message:

Warning: Line 1641, Connect pin 'XYZ' to the default power pg_pin 'VDD'. (LBDB-725)

pg_pin Group

The cell-level pg_pin group is used to model the VDD and VSS pins and virtual VDD and VSS pins of a coarse-grain switch cell. The syntax is based on the Y-2006.06 power and ground pin syntax.

Fine-Grained Switch Support for Macro Cells

A macro cell with a fine-grained switch is a cell that contains a special switch transistor with a control pin that can turn off the power supply of the cell when it is idle. This significantly lowers the power consumption of a design.

With the growing popularity of low-power designs, macro cells with fine-grained switches play an important role. The syntax identifies a cell as a macro cell and specifies the correct power pin that supplies power to each signal pin.

Macro Cell With Fine-Grained Switch Syntax

```
cell(cell_name) {
    is_macro_cell : true;
    switch_cell_type : coarse_grain | fine_grain;

    pg_pin (power/ground pin name) {
        pg_type : primary_power | primary_ground | backup_power | backup_ground;
        direction: input | inout | output;
        ...
    }

    /* This is a special pg pin that uses "switch_function" to describe the logic to shut off the attached design partition */
    pg_pin (internal power/ground pin name) {
        direction: internal | input | output | inout;
        pg_type : internal_power | internal_ground;
        switch_function : "function_string";
        pg_function : "function_string";
        ...
    }

    pin (input_pin_name) {
        direction : input | inout;
        switch_pin : true | false;
    }
}
```

```
    ...
}
...
pin(output_pin_name) {
    direction : output | inout;
    power_down_function : "function_string";
    ...
} /* end pin group */
} /* end cell group */
```

Cell-Level Attributes

The following attributes are cell-level attributes for macro cells with fine-grained switches.

is_macro_cell Attribute

The `is_macro_cell` simple Boolean attribute identifies whether a cell is a macro cell. If the attribute is set to `true`, the cell is a macro cell. If it is set to `false`, the cell is not a macro cell.

switch_cell_type Attribute

The `switch_cell_type` attribute is enhanced to support macro cells with internal switches. The valid enumerated values for this attribute are `coarse_grain` and `fine_grain`.

pg_pin Group

The following attribute can be specified under the `pg_pin` group for macro cells with fine-grained switches.

direction Attribute

The `direction` attribute supports `internal` as a valid value for macros when the internal power and ground is not visible or accessible at the cell boundary.

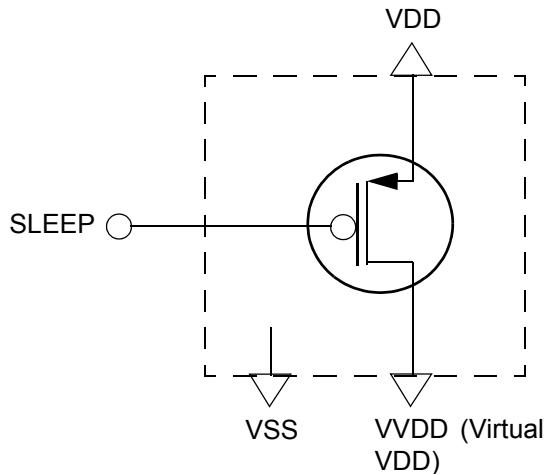
Switch-Cell Modeling Examples

The following sections provide examples for a simple coarse-grain header switch cell and a complex coarse-grain header switch cell.

Simple Coarse-Grain Header Switch Cell

[Figure 13-18](#) and the example that follows it show a simple coarse-grain header switch cell.

Figure 13-18 Simple Coarse-Grain Header Switch Cell



```

library (simple_coarse_grain_lib) {

    ...
    current_unit : 1mA;
    ...

    voltage_map(VDD, 1.0);
    voltage_map(VVDD, 0.8);
    voltage_map(VSS, 0.0);

    operating_conditions(XYZ) {
        process : 1.0;
        voltage : 1.0;
        temperature : 25.0;
    }
    default_operating_conditions : XYZ;

    lu_table_template ( ivt1 ) {
        variable_1 : input_voltage;
        variable_2 : output_voltage;
        index_1 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
        index_2 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
    }
    ...
    cell ( Simple(CG)_Switch ) {
        ...
        switch_cell_type : coarse_grain;

        pg_pin ( VDD ) {
            pg_type : primary_power;
            direction : input;
            voltage_name : VDD;
        }
    }
}

```

```
pg_pin ( VVDD ) {
    pg_type : internal_power;
    voltage_name : VVDD;
    direction : output ;
    switch_function : "SLEEP" ;
    pg_function : "VDD" ;
}

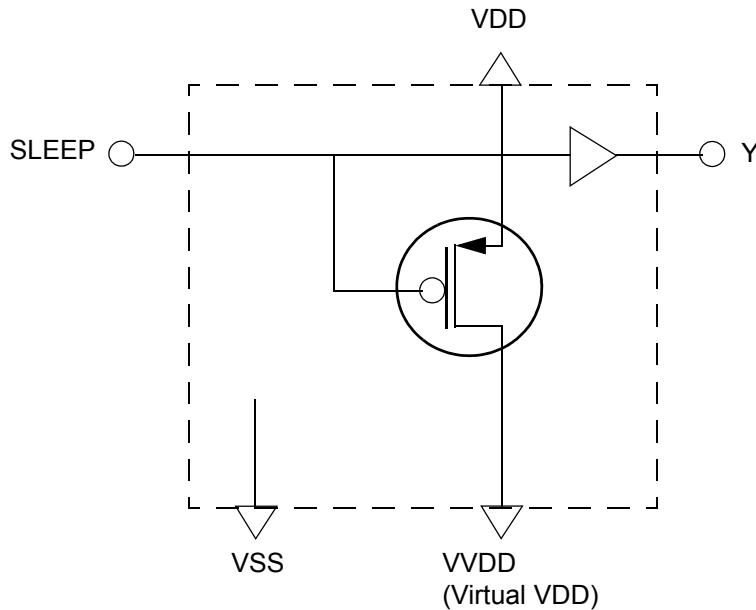
pg_pin ( VSS ) {
    pg_type : primary_ground;
    direction : input;
    voltage_name : VSS;
}

/* I/V curve information */
dc_current ( ivtl ) {
    related_switch_pin : SLEEP;      /* control pin */
    related_pg_pin : VDD;           /* source */
    related_internal_pg_pin : VVDD; /* drain */
    values( "0.010, 0.020, 0.030, 0.030, 0.030", \
            "0.011, 0.021, 0.031, 0.041, 0.051", \
            "0.012, 0.022, 0.032, 0.042, 0.052", \
            "0.013, 0.023, 0.033, 0.043, 0.053", \
            "0.014, 0.024, 0.034, 0.044, 0.054");
}
...
}

pin ( SLEEP ) {
    switch_pin : true;
    capacitance: 1.0;
    related_power_pin : VDD;
    related_ground_pin : VSS;
} /* end pin */
} /* end cell */
} /* end library */
```

Complex Coarse-Grain Header Switch Cell

[Figure 13-19](#) and the example that follows it show a complex coarse-grain header switch cell.

Figure 13-19 Complex Coarse-Grain Header Switch Cell

```

library (complex_coarse_grain_lib) {
    ...
    current_unit : 1mA;
    ...
    voltage_map(VDD, 1.0);
    voltage_map(VVDD, 0.8);
    voltage_map(VSS, 0.0);

    operating_conditions(XYZ) {
        process : 1.0;
        voltage : 1.0;
        temperature : 25.0;
    }
    default_operating_conditions : XYZ;

    lu_table_template ( ivt1 ) {
        variable_1 : input_voltage;
        variable_2 : output_voltage;
        index_1 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
        index_2 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
    }
    ...
    cell ( Complex(CG)_Switch ) {
        ...
        switch_cell_type : coarse_grain;
        pg_pin ( VDD ) {
            pg_type : primary_power;
            voltage_name : VDD;
        }
    }
}

```

```

        direction: input ;
    }
    pg_pin ( VVDD ) {
        pg_type : internal_power;
        direction : output ;
        voltage_name : VVDD;
        switch_function : "SLEEP";
        pg_function : "VDD" ;
    }
    pg_pin ( VSS ) {
        pg_type : primary_ground;
        voltage_name : VSS;
        direction : input ;
    }

/* I/V curve information */
dc_current ( ivt1 ) {
    related_switch_pin : SLEEP;      /* control pin */
    related_pg_pin : VDD;           /* source power pin */
    related_internal_pg_pin : VVDD; /* drain internal power pin*/
    values(      "0.010, 0.020, 0.030, 0.040, 0.050", \
                "0.011, 0.021, 0.031, 0.041, 0.051", \
                "0.012, 0.022, 0.032, 0.042, 0.052", \
                "0.013, 0.023, 0.033, 0.043, 0.053", \
                "0.014, 0.024, 0.034, 0.044, 0.054");
}

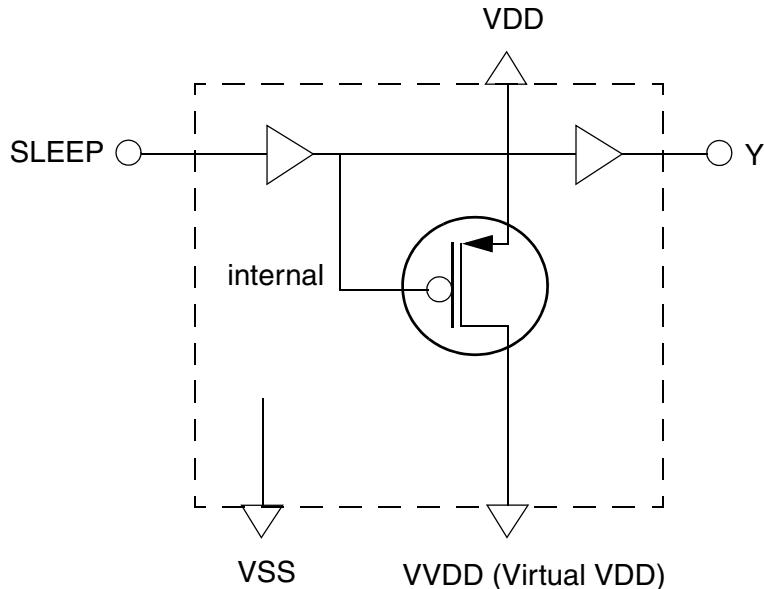
pin ( SLEEP ) {
    direction : input;
    switch_pin : true;
    capacitance: 1.0;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    ...
}
...
pin (Y) {
    direction : output;
    function : "SLEEP";
    related_power_pin : VDD;
    related_ground_pin : VSS;
    power_down_function : "!VDD + VSS";
    timing() {
        related_pin : SLEEP;
        ...
    }
} /* end pin group */
} /* end cell group*/
} /* end library group*/

```

Complex Coarse-Grain Switch Cell With an Internal Switch Pin

[Figure 13-20](#) and the example that follows it show a complex coarse-grain switch cell with an internal switch pin.

Figure 13-20 Complex Coarse-Grain Switch Cell With an Internal Switch Pin



```
library (Complex(CG_lib) {
  ...
  current_unit : 1mA;
  ...

  voltage_map(VDD, 1.0);
  voltage_map(VVDD, 0.8);
  voltage_map(VSS, 0.0);

  operating_conditions(XYZ) {
    process : 1.0;
    voltage : 1.0;
    temperature : 25.0;
  }
  default_operating_conditions : XYZ;

  lu_table_template ( ivt1 ) {
    variable_1 : input_voltage;
    variable_2 : output_voltage;
    index_1 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
    index_2 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
  }
  ...
}
```

```
cell (COMPLEX_HEADER_WITH_INTERNAL_SWITCH_PIN) {
    cell_footprint : complex_mtpmos;
    area : 1.0;
    switch_cell_type : coarse_grain;
    pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
        direction : input;
    }
    pg_pin(VVDD) {
        voltage_name : VVDD;
        pg_type : internal_power;
        direction : output;
        switch_function : "SLEEP" ;
    }
    pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
        direction : input;
    }
}

dc_current(ivt1) {
    related_switch_pin : internal;
    related_pg_pin : VDD;
    related_internal_pg_pin : VVDD;
    values(    "0.010, 0.020, 0.030, 0.040, 0.050", \
              "0.011, 0.021, 0.031, 0.041, 0.051", \
              "0.012, 0.022, 0.032, 0.042, 0.052", \
              "0.013, 0.023, 0.033, 0.043, 0.053", \
              "0.014, 0.024, 0.034, 0.044, 0.054");
}

pin(SLEEP) {
    switch_pin : true;
    direction : input;
    capacitance : 1.0;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    ...
}

pin(Y) {
    direction : output;
    function : "SLEEP";
    related_power_pin : VDD;
    related_ground_pin : VSS;
    power_down_function : "!VDD + VSS";

    timing() {
        related_pin : "SLEEP"
        ...
    }
} /* end pin group */
```

```

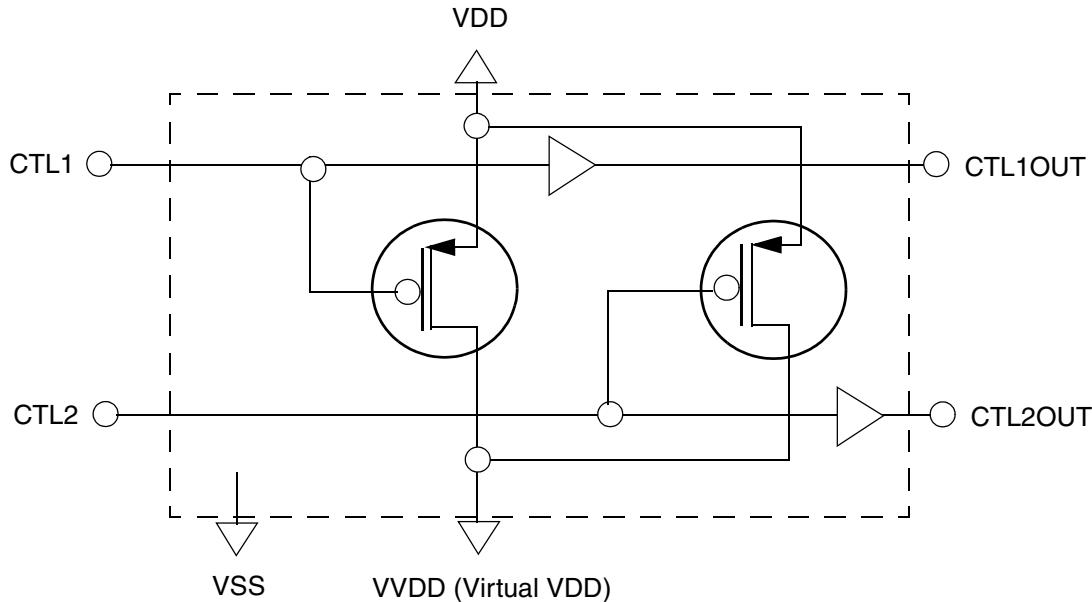
pin(internal) {
    direction : internal;
    timing() {
        related_pin : "SLEEP"
    ...
}
} /* end pin group */
} /* end cell group */
} /* end library group */

```

Complex Coarse-Grain Switch Cell With Parallel Switches

[Figure 13-21](#) and the example that follows it show a complex coarse-grain switch cell with two parallel switches.

Figure 13-21 Complex Coarse-Grain Switch Cell With Two Parallel Switches



```

library (Complex(CG)_lib) {
...
current_unit : 1mA;
...

voltage_map(VDD, 1.0);
voltage_map(VVDD, 0.8);
voltage_map(VSS, 0.0);

operating_conditions(XYZ) {
    process : 1.0;
    voltage : 1.0;
}

```

```

        temperature : 25.0;
    }
default_operating_conditions : XYZ;

lu_table_template ( ivt1 ) {
    variable_1 : input_voltage;
    variable_2 : output_voltage;
    index_1 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
    index_2 ( "0.1, 0.2, 0.4, 0.8, 1.0" );
}
...

cell (COMPLEX_HEADER_WITH_TWO_PARALLEL_SWITCHES) {
    cell_footprint : complex_mtpmos;
    area : 1.0;
    switch_cell_type : coarse_grain;

    pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
        direction : input;
    }
    pg_pin(VVDD) {
        voltage_name : VVDD;
        pg_type : internal_power;
        direction : output;
        switch_function : "CTL1 & CTL2" ;
    }
    pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
        direction : input;
    }
}

dc_current(ivt1) {
    related_switch_pin : CTL1;
    related_pg_pin : VDD;
    related_internal_pg_pin : VVDD;
    values(  "0.010, 0.020, 0.030, 0.040, 0.050", \
             "0.011, 0.021, 0.031, 0.041, 0.051", \
             "0.012, 0.022, 0.032, 0.042, 0.052", \
             "0.013, 0.023, 0.033, 0.043, 0.053", \
             "0.014, 0.024, 0.034, 0.044, 0.054");
}

dc_current(ivt1) {
    related_switch_pin : CTL2;
    related_pg_pin : VDD;
    related_internal_pg_pin : VVDD;
    values(  "0.010, 0.020, 0.030, 0.040, 0.050", \
             "0.011, 0.021, 0.031, 0.041, 0.051", \
             "0.012, 0.022, 0.032, 0.042, 0.052", \
             "0.013, 0.023, 0.033, 0.043, 0.053", \
             "0.014, 0.024, 0.034, 0.044, 0.054");
}

```

```
        "0.014, 0.024, 0.034, 0.044, 0.054");  
    }  
  
    pin(CTL1) {  
        switch_pin : true;  
        direction : input;  
        capacitance : 1.0;  
        related_power_pin : VDD;  
        related_ground_pin : VSS;  
        ...  
    }  
  
    pin(CTL2) {  
        switch_pin : true;  
        direction : input;  
        capacitance : 1.0;  
        related_power_pin : VDD;  
        related_ground_pin : VSS;  
        ...  
    }  
  
    pin(CTL1OUT) {  
        direction : output;  
        function : "CTL1";  
        related_power_pin : VDD;  
        related_ground_pin : VSS;  
        power_down_function : "!VDD + VSS";  
        timing() {  
            related_pin : "CTL1"  
            ...  
        }  
    } /* end pin group */  
    pin(CTL2OUT) {  
        direction : output;  
        function : "CTL1";  
        related_power_pin : VDD;  
        related_ground_pin : VSS;  
        power_down_function : "!VDD + VSS";  
        timing() {  
            related_pin : "CTL2"  
            ...  
        }  
    } /* end pin group */  
} /* end cell group */  
} /* end library group */
```

Switch Cell Library Checks

The Library Compiler tool performs checks for switch cells and issues an error or warning message if certain conditions occur. For a detailed list of library checks for switch cells, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Retention Cell Modeling

Some elements of an electronic design can store the logic state of the design. This is required for the design to wake up in the same state in which it was shut down. The retention cell is one such design element.

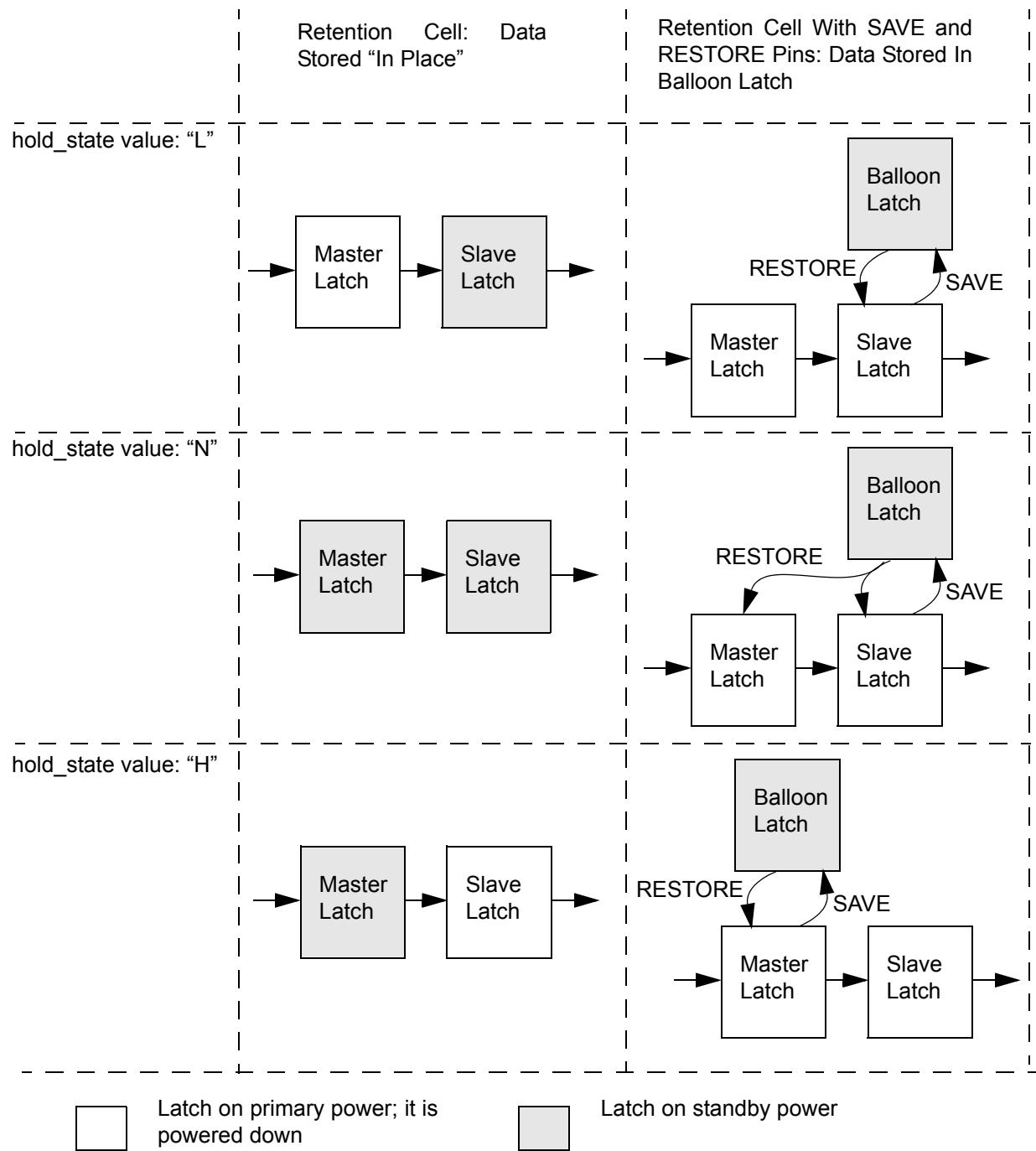
Retention cells are sequential cells that hold their state when the power supply is shut down and restore this state when the power is brought up again. The retention cell or register consists of a main register and a shadow register that has a different power supply. The power supply to the shadow register is always powered on to maintain the memory of the state. Therefore, the shadow register has high threshold-voltage transistors to reduce the leakage power. The main register has low threshold-voltage transistors for performance during the normal operation of the retention cell.

Retention cells are broadly classified into the store-in-place and balloon structures.

[Figure 13-22](#) shows the two main retention cell structures. The store-in-place structure includes a master-slave latch where either the slave or the master latch stores the state of the cell when the master-slave latch is shut down. In this structure, the master-slave latch implements the main register while the latch that stores the state of the cell implements the shadow register. The balloon structure includes a flip-flop or master-slave latch and a balloon latch with a control logic that stores the state of the cell when the master-slave latch is shut down. In this structure, the master-slave latch implements the main register and the balloon latch implements the shadow register. The control logic includes signals, such as save and restore signals that control the data storage in the shadow register and the data transfer between the shadow register and the main register.

Note:

In the Synopsys Low-Power UPF flow, the RTL code for a balloon latch is automatically added to the RTL code for every register that is inferred as a retention cell. This ensures that the register and balloon latch are simulated together. The simulated retention cell model is compliant with the UPF verification flow. This is because, for a particular technology library, formal equivalence checkers compare a retention cell model that consists of a register and balloon latch with the RTL register and balloon latch. Other retention cell models might not be compliant with the UPF verification flow and can cause verification failures.

Figure 13-22 Retention Cell Structures

Modes of Operation

A retention cell has two modes of operation: *normal mode* and *retention mode*. The retention mode has three stages: *save event*, *sleep mode*, and *restore event*. For example, for the balloon structure, the master-slave latch operates in normal mode and the balloon latch operates in retention mode. The master-slave latch is considered to be edge-triggered. The normal and retention modes are described as follows:

- normal mode

In this mode, the retention cell is fully powered on and the master-slave latch operates normally. The balloon latch does not add to the load at the output of the retention cell.

- retention mode

- save event

During this event, the current state of the master-slave latch is saved before the power to the latch is shut down. The balloon latch is considered to be edge-triggered during the save event. The trailing edge of the save control signal is the triggering edge.

- sleep mode

In this mode, the master-slave latch is shut down.

- restore event

During the restore event, a wake-up signal activates the master-slave latch and the data stored in the balloon latch is fed back to the master-slave latch. The state of the master-slave latch is restored just after the power is restored and before the retention cell operation returns to normal mode. The balloon latch is also considered to be edge-triggered during the restore event. The two methods to restore the state are:

- The leading edge of the restore signal is the triggering edge for the balloon latch. The master-slave latch becomes transparent, that is, it does not latch the data. However, it outputs the same state that was saved in the balloon latch.
- The trailing edge is the triggering edge for the balloon latch. The data is fully restored from the balloon latch and the flip-flop starts operating normally. In this method, the restored data is available for a shorter time.

Retention Cell Modeling Syntax

The following syntax shows the modeling of retention cells. The `reference_input` attribute defines the connectivity information for the input pins based on the `reference_pin_names` variable. The `reference_pin_names` variable specifies the internal reference input nodes used within the `ff`, `latch`, `ff_bank`, and `latch_bank` groups.

The sequential components of a retention cell are defined by using the flip-flop and latch syntax. The syntax to model the scan retention cells is identical to the syntax to model the scan sequential components. All scan retention cell functional models have a regular cell function that includes the scan pins as part of the function, while the `test_cell` group models the nonscan functionality of the retention cells with the scan pins that have been specified with the `signal_type` attributes.

```
cell(cell_name) {
    retention_cell : retention_cell_style;

    pg_pin (primary_power_name) {
        voltage_name : primary_power_name;
        pg_type : primary_power;
    }
    pg_pin (primary_ground_name) {
        voltage_name : primary_ground_name;
        pg_type : primary_ground;
    }
    pg_pin (backup_power_name) {
        voltage_name : backup_power_name;
        pg_type : backup_power;
    }
    pg_pin (backup_ground_name) {
        voltage_name : backup_ground_name;
        pg_type : backup_ground;
    }
    pin(pin_name) {
        retention_pin(pin_class, disable_value);
        related_ground_pin : backup_ground;
        related_power_pin : backup_power;
        save_action : L|H|R|F;
        restore_action : L|H|R|F;
        restore_edge_type : edge_trigger | leading | trailing;
        ...
    }
    ...
    retention_condition() {
        power_down_function: "Boolean_function";
        required_condition: "Boolean_function";
    }
    clock_condition() {
        clocked_on : "Boolean_expression";
        required_condition : "Boolean_expression";
        hold_state : L|H|N ;
        clocked_on_also : "Boolean_expression";
        required_condition_also : "Boolean_expression";
        hold_state_also : L|H|N;
    }
    preset_condition() {
        input : "Boolean_expression";
        required_condition : "Boolean_expression" ;
    }
}
```

```

clear_condition() {
    input : "Boolean_expression" ;
    required_condition : "Boolean_expression" ;
}
pin(pin_name) {
    direction : inout | output | internal;
    function : Boolean_equation_with_internal_node_name;
    reference_input : pin_names;
    ...
}
bus(bus_name) {
    bus_type : bus_type_name;
    direction : inout | output | internal;
    function : Boolean_equation_with_internal_node_name;
    reference_input : pin_names;
    ...
}
ff (["reference_pin_names",] variable1, variable2 ) {
    power_down_function : "Boolean_expression" ;
    ...
}
latch (["reference_pin_names",] variable1, variable2 ) {
    power_down_function : "Boolean_expression";
    ...
}
ff_bank (["reference_pin_names",] variable1, variable2, bits) {
    power_down_function : "Boolean_expression";
    ...
}
latch_bank (["reference_pin_names",] variable1, variable2, bits) {
    power_down_function : "Boolean_expression";
    ...
}
statetable (...) {
    power_down_function : "Boolean_expression";
    ...
}
...
}
}

```

Note:

The `-full_table` option of the `report_lib` command reports the state table description of the retention cell in the table format for the normal mode only.

Cell-Level Attributes, Groups, and Variables

This section describes the cell-level attributes, groups, and variables for retention cell modeling.

retention_cell Simple Attribute

The `retention_cell` attribute identifies the type of the retention cell or register. For a given cell, there can be multiple types of retention cells that have the same function in normal mode but different sleep signals, wake signals, or clocking schemes. For example, if a D flip-flop supports two retention strategies, such as `type1` (where the data is transferred when the clock is low) and `type2` (where the data is transferred when the clock is high), the values of the `retention_cell` attribute are different, such as `DFF_type1` and `DFF_type2`.

ff, latch, ff_bank, and latch_bank Groups

The `ff`, `latch`, `ff_bank`, and `latch_bank` groups define sequential blocks. Define these groups at the cell level. You can specify one or more of these groups within a `cell` group.

retention_condition Group

The `retention_condition` group is a group of attributes that specify the conditions for the retention cell to hold its state during the retention mode. The `retention_condition` group includes the `power_down_function` and `required_condition` attributes.

power_down_function Attribute

The `power_down_function` attribute specifies the Boolean condition for the retention cell to be powered down—that is, the primary power to the cell is shut down. When this Boolean condition evaluates to true, it triggers the evaluation of the control input conditions specified by the `required_condition` attribute.

required_condition Attribute

The `required_condition` attribute specifies the control input conditions during the retention mode. For example, in [Figure 13-27 on page 13-108](#), the retention signal, RET, is low during the retention mode. These conditions are checked when the Boolean condition specified by the `power_down_function` attribute evaluates to true. If these conditions are not met, the cell is considered to be in an illegal state.

Note:

Within the `retention_condition` group, the `power_down_function` attribute by itself does not specify the retention mode of the cell. The conditions specified by the `required_condition` attribute ensure that the retention control pin is in the correct state when the primary power to the cell is shut down.

clock_condition Group

The `clock_condition` group is a group of attributes that specify the conditions for correct signal during clock-based events. The `clock_condition` group includes two classes of attributes: attributes without the `_also` suffix and attributes with the `_also` suffix. These are similar to the `clocked_on` and `clocked_on_also` attributes of the `ff` group.

clocked_on and clocked_on_also Attributes

The `clocked_on` and `clocked_on_also` attributes specify the active edge of the clock signal. The Boolean expression of the `clocked_on` attribute must be identical to the one specified in the `clocked_on` attribute of the corresponding `ff` or `ff_bank` group.

For example, for a master-slave latch, use the `clocked_on` attribute on the clock to the master latch and the `clocked_on_also` attribute on the clock to the slave latch.

Note:

A single-stage flip-flop or latch does not use the `clocked_on_also` attribute.

required_condition and required_condition_also Attributes

The `required_condition` and `required_condition_also` attributes specify the input conditions during the active edge of the clock signal. These conditions are checked, respectively, at the values specified by the `clocked_on` and `clocked_on_also` attributes. If any one of the conditions are not met, the cell is considered to be in an illegal state.

For example, when the `required_condition` attribute is checked at the rising edge of the clock signal specified by the `clocked_on` attribute, the `required_condition_also` attribute is also checked at the rising edge of the clock signal specified by the `clocked_on_also` attribute. If the `clocked_on_also` attribute is not specified, the `required_condition_also` attribute is checked at the falling edge of the clock signal specified by the `clocked_on` attribute. This condition is checked when the slave latch is in the hold state.

hold_state and hold_state_also Attributes

The `hold_state` and `hold_state_also` attributes specify the values for the Boolean expressions of the `clocked_on` and `clocked_on_also` attributes during the retention mode. The valid values are `L` (low), `H` (high), or `N` (no change).

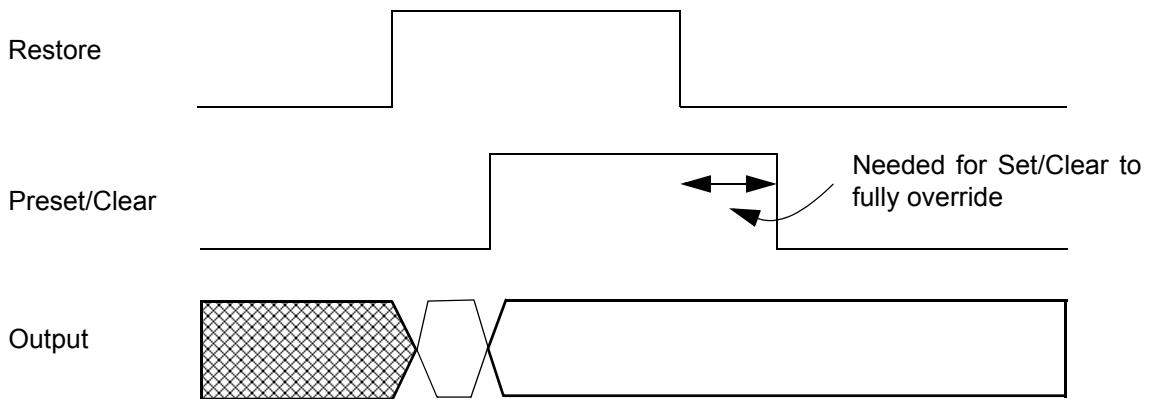
If the data is restored to both the master and slave latches, the value of the `hold_state` attribute is `N`. If the data is restored to the slave latch, the value of the `hold_state` attribute is `L`.

preset_condition and clear_condition Groups

The `preset_condition` and `clear_condition` groups contain attributes that specify the conditions, respectively, for the present and clear signals in the normal mode of the retention cell.

The asynchronous control signals, preset and clear, have higher priority over the clock signal. However, these signals do not control the balloon latch. Therefore, if the preset or clear signals are asserted during the restore event, these signals need to be active for a time longer than the restore event so that the master-slave latch content is successfully overwritten, as shown in [Figure 13-23](#). Therefore, the preset and clear signals must be checked at the trailing edge.

Figure 13-23 Preset and Clear Signal Overlaps With Restore



input Attribute

The `input` attribute specifies how the preset or clear control signal is asserted. The Boolean expression of the `input` attribute must be identical to one specified in the `input` attribute of the `ff` group.

required_condition Attribute

The `required_condition` attribute specifies the input condition during the active edge of the preset or clear signal. The `required_condition` attribute is checked at the trailing edge of the preset or clear signal. When the input condition is not met, the cell is in an illegal state.

reference_pin_names Variable

The `reference_pin_names` variable specifies the input nodes that are used for internal reference within the `ff`, `latch`, `ff_bank`, or `latch_bank` groups. If you do not specify the `reference_pin_names` variable, the node names in the `ff`, `latch`, `ff_bank`, or `latch_bank` groups are considered to be the actual pin or bus names of the cell.

variable1 and variable2 Variables

The `variable1` and `variable2` variables define the output nodes for internal reference. The values of the `variable1` and `variable2` variables in the `ff`, `latch`, `ff_bank`, or `latch_bank` groups must be unique for a cell.

bits Variable

The `bits` variable defines the width of the `ff_bank` and `latch_bank` component.

Pin-Level Attributes

This section describes the pin-level attributes for retention cell modeling.

retention_pin Complex Attribute

The `retention_pin` attribute identifies the retention pins of a retention cell. In the normal mode, the retention pins are disabled.

The `retention_pin` attribute has the following arguments:

- Pin class

The values are

restore

Restores the state of the cell.

save

Saves the state of the cell.

save_restore

Saves and restores the state of the cell. When a single pin in a retention cell performs both the `save` and `restore` operations, specify the `retention_pin` attribute with the `save_restore` value. The retention pin is in `save` mode when it saves the data. The retention pin is in `restore` mode when it restores the data.

- Disable value

Defines the value of the retention pin when the cell works in normal mode. The valid values are 0 and 1.

In the following example, the disable value of the `save_restore` pin, `S_R`, is 1. The retention cell works in `save` mode when the disable value of the `save_restore` pin is 1, and in `restore` mode when the disable value is 0.

```

pin(S_R) {
    ...
    retention_pin (save_restore, 1);
    ...
} /* end pin group */

```

Note:

If the retention signal pin does not have the `related_power_pin` and `related_ground_pin` attributes, that is, it is not associated with a power and ground pin, the Library Compiler tool associates this pin to the first specified primary power and ground rails that are defined on the cell, and it marks the cell with the `dont_touch` or `dont_use (d,u)` attribute upon generating the following warning messages:

Warning: Line 20, Cell 'retention_cell', pin 'RET', Connect pin 'RET' to the default power pg_pin 'VDD'. (LBDB-725)
 Warning: Line 20, Cell 'retention_cell', pin 'RET', Connect pin 'RET' to the default ground pg_pin 'VSS'. (LBDB-725)
 Warning: Line 20, Cell 'retention_cell', pin 'RET', is missing related_power_pin to a 'power' pg_pin, so it becomes a black box cell for multivoltage functional optimization flow. It is being marked as `dont_touch`, `dont_use`. (LBDB-914)
 Warning: Line 100, Cell 'retention_cell', pin 'RET', is missing related_ground_pin to a 'ground' pg_pin, so it becomes a black box cell for multivoltage functional optimization flow. It is being marked `dont_touch`, `dont_use`. (LBDB-915)

function Attribute

The `function` attribute maps an output, inout, or internal pin to a corresponding internal node or a value of the `variable1` or `variable2` variable in a `ff`, `latch`, `ff_bank`, or `latch_bank` group. The `function` attribute also accepts a Boolean equation containing the `variable1` or `variable2` variable and other input, inout, or internal pins. Define the `function` attribute in a `pin` or `bus` group.

reference_input Attribute

The `reference_input` attribute specifies the input pins that map to the reference pin names of the corresponding `ff`, `latch`, `ff_bank`, or `latch_bank` group. For each inout, output, or internal pin, the `variable1` or `variable2` value specified in the `function` statement determines the corresponding `ff`, `latch`, `ff_bank`, or `latch_bank` group. You can define the `reference_input` attribute in a `pin` or `bus` group.

save_action and restore_action Attributes

The `save_action` and `restore_action` attributes specify where the save and restore events occur with respect to the save and restore control signals, respectively. Valid values are `L` (low), `H` (high), `R` (rise), and `F` (fall). The `L` or `H` values indicate that the data is actually

stored in the balloon latch at the trailing edge of the save signal. The R or F values indicate that this edge of the restore signal specifies when the data stored in the balloon latch is available at the output of the master-slave latch.

restore_edge_type Attribute

The `restore_edge_type` attribute specifies the type of the edge of the restore signal where the output of the master-slave latch is restored. The `restore_edge_type` attribute supports the following edge types: `edge_trigger`, `leading`, and `trailing`. The default edge type is `leading`. This is because the other control signals, such as clock, preset, and clear are of `leading` edge type, that is, they make the data available at the output of the master-slave latch when the latch is transparent.

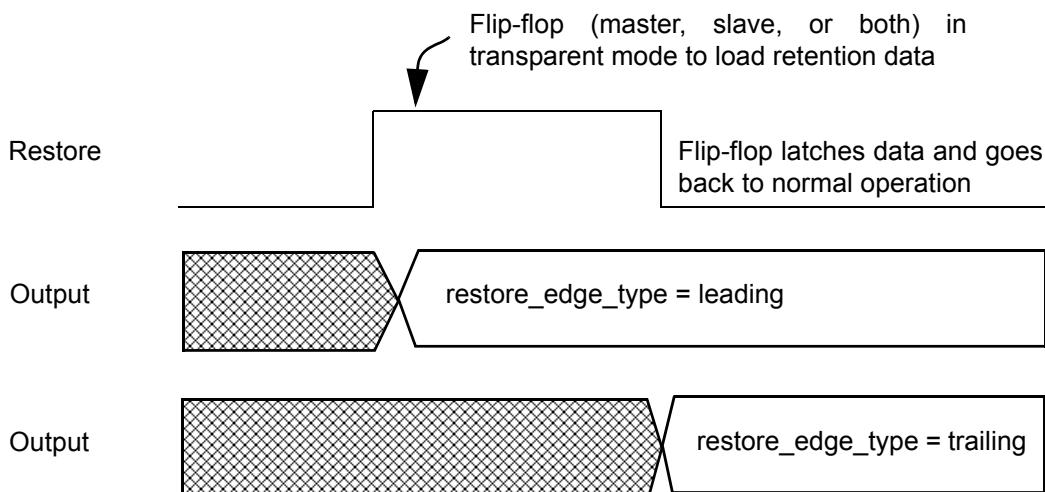
The `edge_trigger` type indicates that the output of the master-slave latch is restored at the edge of the restore signal. The master-slave latch resumes normal operation immediately thereafter.

The `leading` edge type indicates that the output of the master-slave latch is restored at the leading edge of the restore signal. The master-slave latch resumes normal operation after the trailing edge of the restore signal.

The `trailing` edge type indicates that the output of the master-slave latch is restored at the trailing edge of the restore signal. The master-slave latch resumes normal operation after the trailing edge of the restore signal.

Figure 13-24 shows the valid data windows when the `restore_edge_type` attribute is set to the `leading` and `trailing` edge types.

Figure 13-24 Valid Data Window for leading and trailing Values of the restore_edge_type Attribute



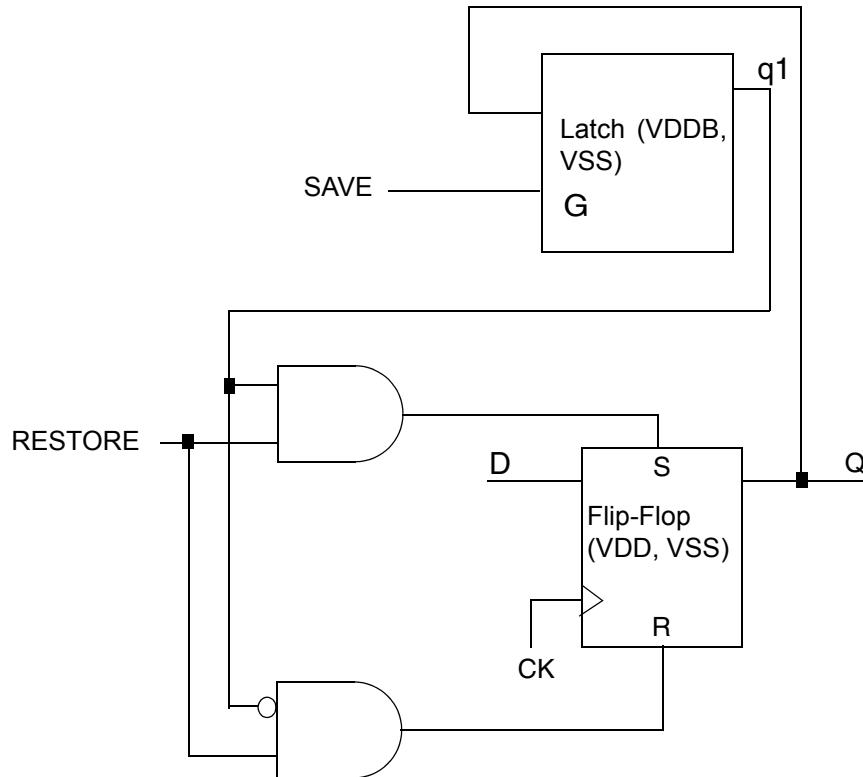
save_condition and restore_condition Attributes

The `save_condition` and `restore_condition` attributes specify the input conditions during the save and restore events respectively. These conditions are respectively checked at the values of the `save_action` and `restore_action` attributes. If any one of the conditions are not met, the cell is considered to be in an illegal state.

Retention Cell Model Examples

[Figure 13-25](#) shows a retention cell structure that is defined using the `ff` or `latch` group. The cell has a balloon latch structure to store the data when the main flip-flop is shut down. The data is transferred to the output of the retention cell at the rising edge of the clock signal, CK. The SAVE and RESTORE pins save and restore the data.

Figure 13-25 Retention Cell Model Schematic With Balloon Latch



Example 13-11 Retention Cell With Balloon Latch

```
library (BALLOON_RET_FLOP) {
  delay_model : table_lookup;
  input_threshold_pct_rise : 50 ;
  input_threshold_pct_fall : 50 ;
```

```
output_threshold_pct_rise : 50 ;
output_threshold_pct_fall : 50 ;
slew_lower_threshold_pct_fall : 30.0 ;
slew_lower_threshold_pct_rise : 30.0 ;
slew_upper_threshold_pct_fall : 70.0 ;
slew_upper_threshold_pct_rise : 70.0 ;

time_unit : "1ns";
voltage_unit : "1V";
current_unit : "1uA";
pulling_resistance_unit : "1kohm";
capacitive_load_unit (0.1,ff);

voltage_map (VDD, 1.0);
voltage_map (VDDB, 0.9);
voltage_map (VSS, 0.0);

nom_process : 1.0;
nom_temperature : 25.0;
nom_voltage : 1.1;

operating_conditions(typ) {
    process : 1.0 ;
    temperature : 25 ;
    voltage : 1.1 ;
    tree_type : "balanced_tree" ;
}
default_operating_conditions : typ;
wire_load("05*05") {
    resistance : 1.0 ;
    capacitance : 25 ;
    area : 1.1 ;
    slope : "balanced_tree" ;
    fanout_length(1,0.39);
}

cell (balloon_ret_cell) {

    retention_cell : retdiff;
    area : 1.0 ;

    pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
    }
    pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
    }
    pg_pin(VDDB) {
        voltage_name : VDDB;
        pg_type : backup_power;
    }
}
```

```

ff ( Q1, QN1 ) {
    clocked_on : " CK ";
    next_state : " D ";
    clear : " RESTORE * !Q2 ";
    preset : " RESTORE * Q2 ";
    clear_preset_var1 : "L";
    clear_preset_var2 : "H";
    power_down_function : "!VDD+VSS";
/* Latch 1 is powered by primary power supply */
}
latch("Q2", "QN2") {
    enable : " SAVE ";
    data_in : "Q";
    power_down_function : "!VDDB+VSS";
/* Latch 2 is powered by backup power supply */
}
clock_condition() {
    clocked_on : "CK";
    required_condition : "RESTORE";
    hold_state : L;
}

pin(RESTORE) {
    direction : input;
    capacitance : 0.1;
    related_power_pin : VDDB;
    related_ground_pin : VSS;
    retention_pin	restore, "0");
    restore_action : "H";
    restore_condition : "!CK";
    restore_edge_type : "leading";
}
pin(SAVE) {
    direction : input;
    capacitance : 0.1;
    related_power_pin : VDDB;
    related_ground_pin : VSS;
    retention_pin(save, "0");
    save_action : "H";
    save_condition : "!CK";
}
pin(D) {
    direction : input;
    capacitance : 0.1;
    related_power_pin : VDD;
    related_ground_pin : VSS;
}
pin(CK) {
    direction : input;
    clock : true;
    capacitance : 0.1;
}

```

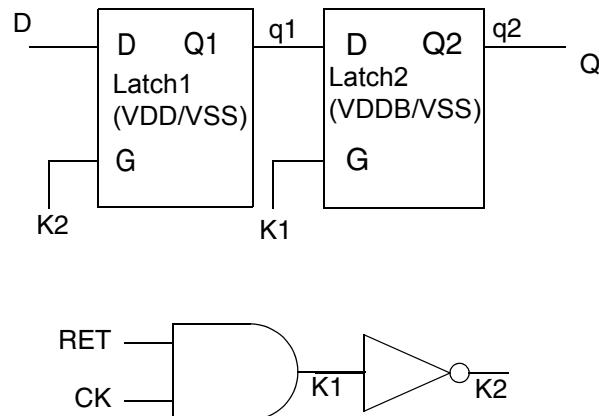
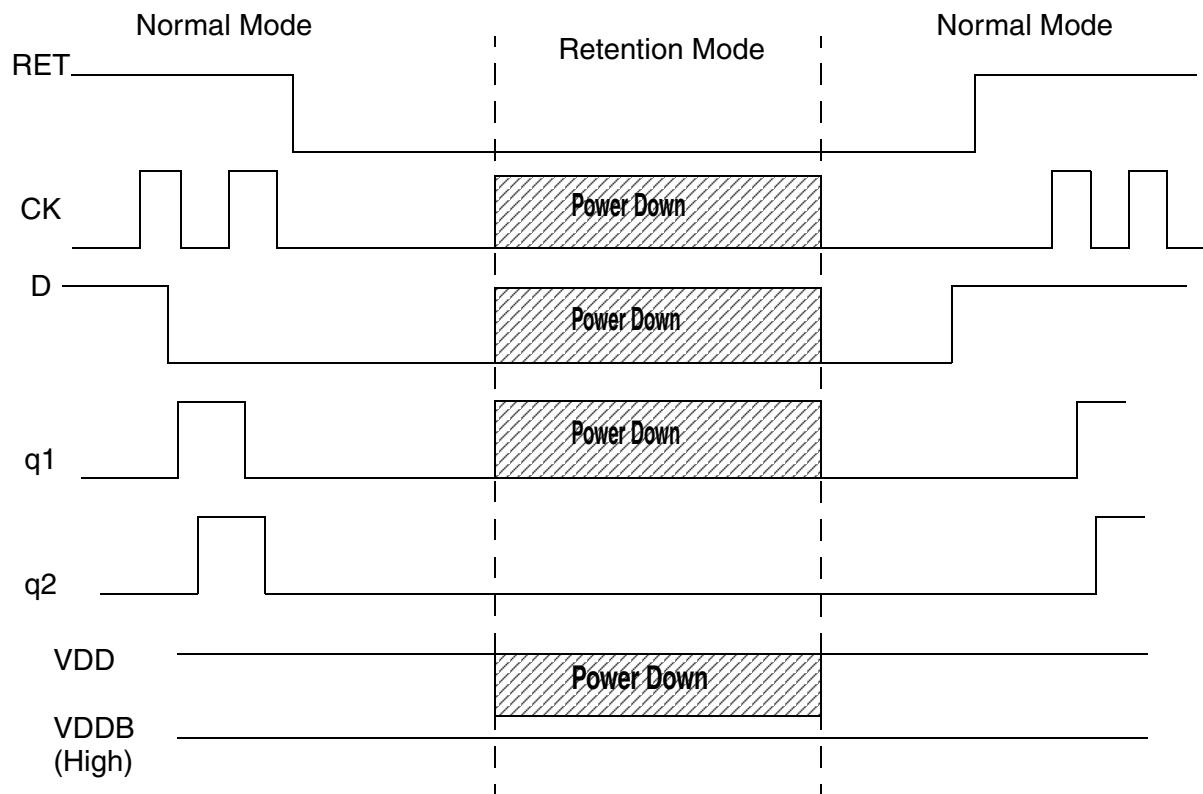
```

        related_power_pin : VDD;
        related_ground_pin : VSS;
    }
    pin(Q) {
        direction : output;
        function : "Q1";
        related_power_pin : VDD;
        related_ground_pin : VSS;
        timing() {
            related_pin : "CK";
            timing_type : rising_edge;
            cell_rise(scalar) { values ( "0.1"); }
            rise_transition(scalar) { values ( "0.1"); }
            cell_fall(scalar) { values ( "0.1"); }
            fall_transition(scalar) { values ( "0.1"); }
        }
    }
    retention_condition() {
        power_down_function : "!VDD+VSS";
        required_condition: "!SAVE";
    }
/*
pin(q1) {
    direction : internal;
    function : "Q2";
}
*/
} /* End cell group */
} /* End Library group */

```

[Figure 13-26](#) shows a schematic of a basic retention cell. The state of the cell is stored inside the slave latch that is powered by the backup power VDDB. [Figure 13-27](#) shows the valid values of the input control signals when the cell is in retention mode.

In the figure, and in [Example 13-12](#), the reference cells are defined by using the `latch` group syntax. The connectivity information for the input pins is specified in the `reference_input` attribute that is mapped to the reference pin names of the corresponding latch group.

Figure 13-26 Simple Retention Cell Model Schematic*Figure 13-27 Valid RET and CK Signal Values in the Retention Mode*

Note:

To retain the stored data, the retention signal, RET, must be low during the retention mode. This condition is specified by the `required_condition` attribute of the `retention_condition` group.

Example 13-12 Retention Cell Model Example Using Multiple Latch Group

```

library (RET_FLOP) {
delay_model : table_lookup;

input_threshold_pct_rise      : 50 ;
input_threshold_pct_fall       : 50 ;
output_threshold_pct_rise     : 50 ;
output_threshold_pct_fall     : 50 ;
slew_lower_threshold_pct_fall : 30.0 ;
slew_lower_threshold_pct_rise : 30.0 ;
slew_upper_threshold_pct_fall : 70.0 ;
slew_upper_threshold_pct_rise : 70.0 ;

time_unit                      : "1ns";
voltage_unit                    : "1V";
current_unit                    : "1uA";
pulling_resistance_unit        : "1kohm";
capacitive_load_unit (0.1,ff);

voltage_map (VDD, 1.0);
voltage_map (VDDB, 0.9);
voltage_map (VSS, 0.0);

nom_process                     : 1.0;
nom_temperature                 : 25.0;
nom_voltage                     : 1.1;

operating_conditions(xyz) {
    process : 1.0 ;
    temperature : 25 ;
    voltage : 1.1 ;
    tree_type : "balanced_tree" ;
}
default_operating_conditions : xyz;
... /* Other library level attributes and groups */

cell (retention_flip_flop) {
    retention_cell : retdiff;
    area : 1.0;

    pg_pin (VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
    }
    pg_pin (VSS) {
        voltage_name : VSS;
    }
}

```

```
    pg_type : primary_ground;
}
pg_pin (VDDB) {
    voltage_name : VDDB;
    pg_type : backup_power;
}
latch ( Q1, QN1 ) {
    enable : "(RET * CK)'";
    data_in : "D";
    power_down_function : "!VDD+VSS";
/* Latch1 is powered by primary power supply */
}
latch("Q2", "QN2") {
    enable : "RET * CK";
    data_in : "Q1";
    power_down_function : "!VDDB+VSS";
/* Latch2 is powered by backup power supply */
}
clock_condition() {
    clocked_on : "CK";
    required_condition : "RET";
    hold_state : "L";

pin (RET) {
    direction : input;
    capacitance : 0.1;
    related_power_pin : VDDB;
    related_ground_pin : VSS;
    retention_pin(save_restore, "1");
    save_action : "L";
    restore_action : "H";
    save_condition : "!CK";
    restore_condition : "!CK";
    restore_edge_type : "leading";
}
pin(D) {
    direction : input;
    capacitance : 0.1;
    related_power_pin : VDD;
    related_ground_pin : VSS;
}
pin(CK) {
    direction : input;
    clock : true;
    capacitance : 0.1;
    related_power_pin : VDD;
    related_ground_pin : VSS;
}
pin (Q) {
    direction : output;
    function : "Q2";
    related_power_pin : VDD;
    related_ground_pin : VSS;
```

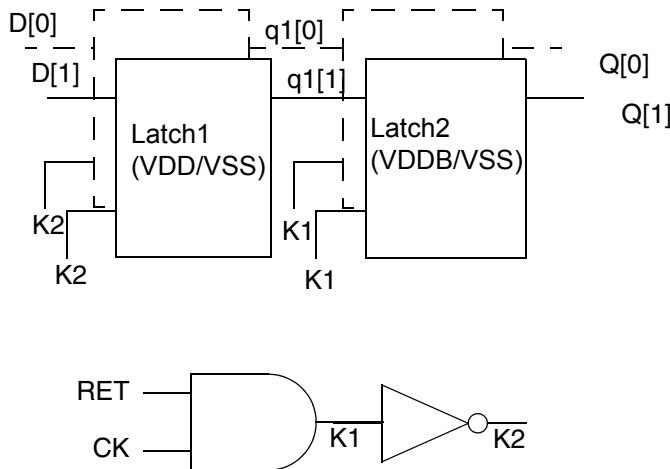
```

        timing() {
            related_pin : "CK";
            timing_type : rising_edge;
            cell_rise(scalar) { values ( "0.1"); }
            rise_transition(scalar) { values ( "0.1"); }
            cell_fall(scalar) { values ( "0.1"); }
            fall_transition(scalar) { values ( "0.1"); }
        }
    }
    retention_condition() {
        power_down_function : "!VDD+VSS";
        required_condition: "!RET";
    }
} /* End Cell group */
} /* End Library group */

```

[Figure 13-28](#) and [Example 13-13](#) show a retention cell structure that is defined using the `latch_bank` group that has multibit parallel inputs and output buses in the datapath and the clock path.

Figure 13-28 Multibit (2-bit) Retention Cell Model Schematic



Example 13-13 Retention Cell Model Example Using Multiple latch_bank Groups

```

library (RET_FLOP_BANK) {

    input_threshold_pct_rise : 50 ;
    input_threshold_pct_fall : 50 ;
    output_threshold_pct_rise : 50 ;
    output_threshold_pct_fall : 50 ;
    slew_lower_threshold_pct_fall : 30.0 ;
    slew_lower_threshold_pct_rise : 30.0 ;
    slew_upper_threshold_pct_fall : 70.0 ;
    slew_upper_threshold_pct_rise : 70.0 ;
}

```

```
time_unit : "1ns";
voltage_unit : "1V";
current_unit : "1uA";
pulling_resistance_unit : "1kohm";
capacitive_load_unit (0.1,ff);

voltage_map (VDD, 1.0);
voltage_map (VDDB, 0.9);
voltage_map (VSS, 0.0);

nom_process : 1.0;
nom_temperature : 25.0;
nom_voltage : 1.1;

operating_conditions(xyz) {
    process : 1.0 ;
    temperature : 25 ;
    voltage : 1.1 ;
    tree_type : "balanced_tree" ;
}
default_operating_conditions : xyz;

type (bus2) {
    base_type : array;
    data_type : bit;
    bit_width : 2;
    bit_from : 0;
    bit_to : 1;
    downto : false;
}

cell (retention_flip_bank) {
    retention_cell : retdiff_bank;
    area : 1.0;

    pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
    }

    pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
    }

    pg_pin(VDDB) {
        voltage_name : VDDB;
        pg_type : backup_power;
    }

    latch_bank ("Q1", "QN1", 2) {
        enable : "(RET * CK)'";
        data_in : "SE'*D + SE*SI";
    }
}
```

```
    power_down_function : "!VDD+VSS";
}
latch_bank ("Q2", "QN2", 2) {
    enable : " RET * CK ";
    data_in : " q1 ";
    power_down_function : "!VDDB+VSS";
}
clock_condition() {
    clocked_on : " CK ";
    required_condition : " RET ";
    hold-state : " L ";
}

bus(D) {
    bus_type : bus2;
    direction : input;
    capacitance : 0.1;
    related_power_pin : VDD;
    related_ground_pin : VSS;
}
bus(q1) {
    bus_type : bus2;
    direction : internal;
    function : "Q1";
    related_power_pin : VDD;
    related_ground_pin : VSS;
}
bus(Q) {
    bus_type : bus2;
    direction : output;
    function : "Q2";
    related_power_pin : VDD;
    related_ground_pin : VSS;
}
pin(RET) {
    direction : input;
    capacitance : 0.1;
    related_power_pin : VDDB;
    related_ground_pin : VSS;
    retention_pin("restore", 1);
    save_action : "L";
    restore_action : "H";
    save_condition : "!CK";
    restore_condition : "!CK";
    restore_edge_type : "leading";
}

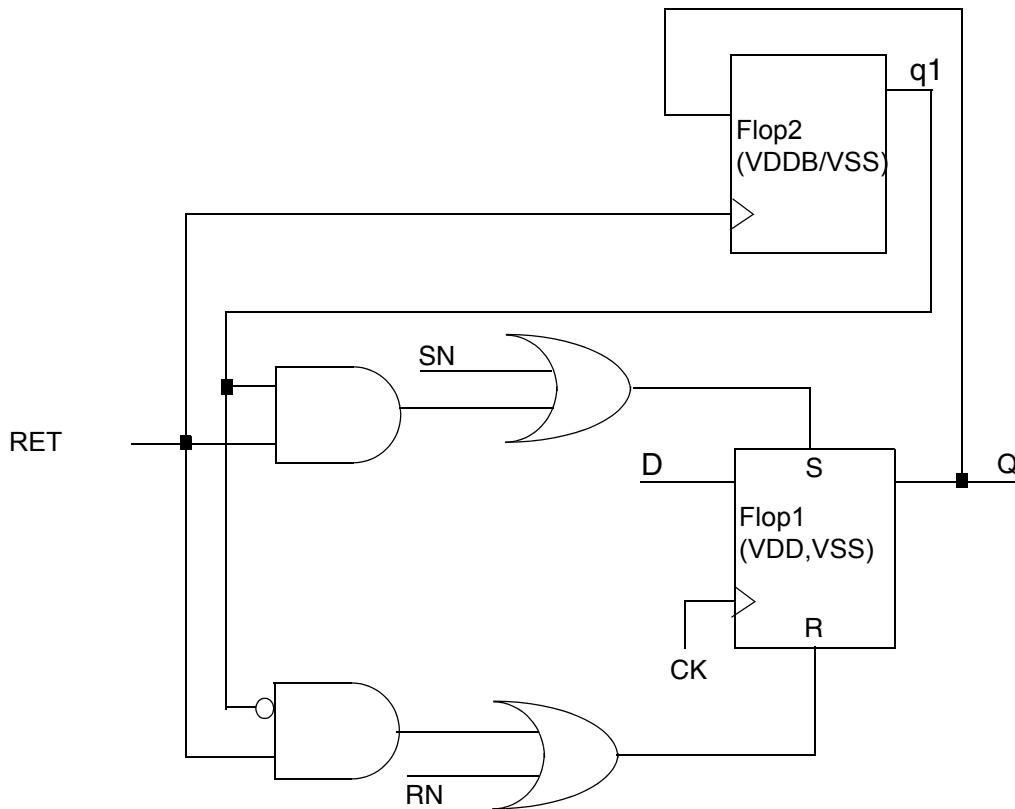
pin(CK) {
    direction : input;
    clock : true;
    capacitance : 0.1;
    related_power_pin : VDD;
    related_ground_pin : VSS;
}
```

```
retention_condition() {
    power_down_function : "!VDD+VSS";
    required_condition: "!RET";
}
bus(SI) {
    bus_type : bus2;
    direction : input;
    clock : true;
    capacitance : 0.1;
    related_power_pin : VDD;
    related_ground_pin : VSS;
}
bus(SE) {
    bus_type : bus2;
    direction : input;
    clock : true;
    capacitance : 0.1;
    related_power_pin : VDD;
    related_ground_pin : VSS;
}
cell_leakage_power : 1.000000;

} /* End Cell group */
} /* End Library group */
```

Figure 13-29 shows a retention cell structure with two `ff` groups. The cell has a balloon latch or flip-flop to store the data when the main flip-flop is shut down. The data is transferred to the output of the main flip-flop, `Flop1`, on the rising edge of the clock signal, `CK`, when the asynchronous preset, `SN`, and clear, `RN`, are inactive. In retention mode, the retention pin, `RET`, saves the data into the balloon flip-flop and restores the data from the balloon flip-flop to the output pin of the retention cell. At the rising edge of the `RET` signal, the data is saved inside the balloon flip-flop, `Flop2`, and `Flop1` is shut down. When the power to `Flop1` is brought up and the retention pin, `RET`, becomes inactive, the data from `Flop2` is restored to `Flop1` at the falling edge of the `RET` signal.

Figure 13-29 Edge-Triggered Retention Cell Model Schematic



Example 13-14 Retention Cell With Edge-Triggered Balloon Logic

```
library(edge_triggered_retention) {
  delay_model : table_lookup;
  time_unit           : "1ns";
  voltage_unit        : "1V";
  current_unit        : "1uA";
  capacitive_load_unit(0.1,ff);
  default_fanout_load : 1.0;
  default inout_pin_cap : 1.0;
  default input_pin_cap : 1.0;
  default output_pin_cap : 1.0;
  input_threshold_pct_rise   : 50 ;
  input_threshold_pct_fall   : 50 ;
  output_threshold_pct_rise  : 50 ;
  output_threshold_pct_fall  : 50 ;
  slew_lower_threshold_pct_fall : 30.0 ;
  slew_lower_threshold_pct_rise : 30.0 ;
  slew_upper_threshold_pct_fall : 70.0 ;
  slew_upper_threshold_pct_rise : 70.0 ;
  voltage_map(VDD, 1.0);
  voltage_map(VDDB, 1.0);
```

```

voltage_map(VSS, 0.0);
  nom_process : 1.0;
  nom_temperature : 25.0;
  nom_voltage : 1.1;
operating_conditions(xyz) {
  process : 1.0 ;
  temperature : 25 ;
  voltage : 1.1 ;
  tree_type : "balanced_tree" ;

}
default_operating_conditions : xyz;
cell (edge_trigger) {
  retention_cell : "edge_trigger";
  ... /* Other cell-level attributes and groups */
pg_pin (VDD) {
  voltage_name : "VDD";
  pg_type : "primary_power";
}
pg_pin (VDDB) {
  voltage_name : "VDDB";
  pg_type : "backup_power";
}
pg_pin (VSS) {
  voltage_name : "VSS";
  pg_type : "primary_ground";
}
ff ("IQ1" , "IQN1") {
  next_state : "D";
  clocked_on : "CK";
  clear : "RN + (RET * q1')";
  preset : "SN + (RET * q1)";
  clear_preset_var1 : L;
  clear_preset_var1 : H;
  power_down_function : "!VDD + VSS"; /* Flip-Flop "Flop1" is powered
by Primary power supply */
}
ff ("IQ2" , "IQN2") {
  next_state : "Q";
  clocked_on : "RET";
  power_down_function : "!VDDS + VSS"; /* Flip-Flop "Flop2" is powered
by Primary power supply */
}
clock_condition() {
  clocked_on : "CK"; /* clock must be Low to go into retention mode */
  hold_state : "N"; /* when clock switches (either direction), RET must
be High */
  condition : "RET";
}
clear_condition() {
  input : "!RN"; /* When clear de-asserts, RET must be high to allow
Low value to be transferred to Flop1 */
}

```

```

        required_condition : "RET";
    }
    preset_condition() {
        input : "!SN"; /* When clear de-asserts, RET must be high to allow
High value to be transferred to Flop1 */
        required_condition : "RET";
    }
    retention_condition() {
        power_down_function : "!VDD+VSS";
        required_condition: "!RET";
    }
    pin (q1) {
        direction : "internal";
        function : "IQ2";
        related_power_pin : "VDD";
        related_ground_pin : "VSS";
        ... /* Other pin-level attributes and groups */
    }
    pin (CK) {
        direction : "input";
        clock : true;
        capacitance : 1.0;
        related_power_pin : "VDD";
        related_ground_pin : "VSS";
        ... /* Other pin-level attributes and groups */
    }
    pin (RET) {
        related_power_pin : "VDD";
        related_ground_pin : "VSS";
        capacitance : 1.0;
        direction : "input";
        retention_pin("save_restore",0);
        save_action : "R";
        restore_action : "R";
        save_condition : "!CK";
        restore_condition : "!CK";
        restore_edge_type : "leading";
        ... /* Other pin-level attributes and groups */
    }
    pin (D) {
        direction : "input";
        related_power_pin : "VDD";
        related_ground_pin : "VSS";
        capacitance : 1.0;
        ... /* Other pin-level attributes and groups */
    }
    pin (RN) {
        direction : "input";
        related_power_pin : "VDD";
        related_ground_pin : "VSS";
        capacitance : 1.0;
        ... /* Other pin-level attributes and groups */
    }

```

```

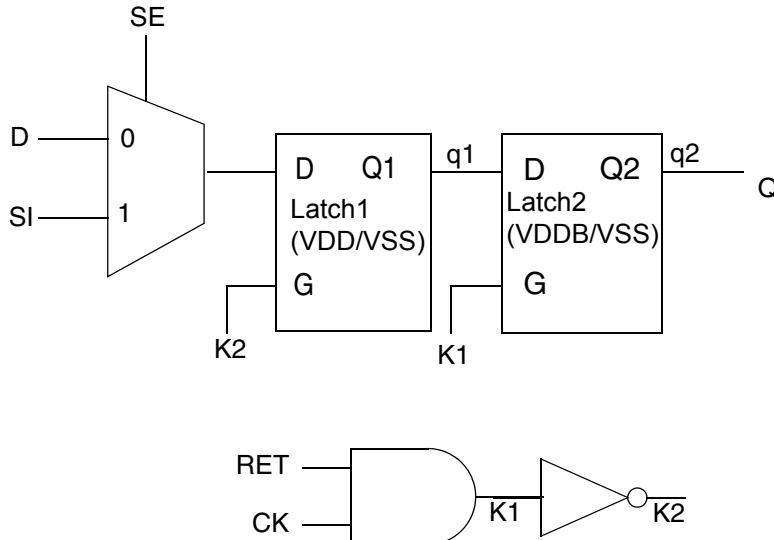
}
pin (SN) {
    direction : "input";
    related_power_pin : "VDD";

    related_ground_pin : "VSS";
    capacitance : 1.0;
    ... /* Other pin-level attributes and groups */
}
pin (Q) {
    direction : "output";
    function : "IQ1";
    related_power_pin : "VDD";
    related_ground_pin : "VSS";
    ... /* Other pin-level attributes and groups */
} /* End Pin group */
} /* End Cell group */
} /* End Library group */

```

[Figure 13-30](#) and [Example 13-15](#) show a scan retention cell structure that is defined using the `latch` group. The cell is the scan version of the retention cell modeled in [Example 13-12](#). The `test_cell` group includes only the function of the nonscan version of the retention cell. The Library Compiler tool can support multiple `latch`, `ff`, `latch_bank`, `ff_bank` groups in the `test_cell` group. Similar to [Example 13-12](#), this retention cell structure is also a store in-place retention cell.

Figure 13-30 MUX-Scan Retention Cell Model Schematic



Example 13-15 MUX-Scan Retention Cell Model

```

library (Retention_cell_Example) {
delay_model : table_lookup;
time_unit           : "1ns";

```

```

voltage_unit : "1V";
current_unit : "1uA";
capacitive_load_unit (0.1,ff);
default_fanout_load : 1.0;
default inout_pin_cap : 1.0;
default input_pin_cap : 1.0;
default output_pin_cap : 1.0;
input_threshold_pct_rise : 50 ;
input_threshold_pct_fall : 50 ;
output_threshold_pct_rise : 50 ;
output_threshold_pct_fall : 50 ;
slew_lower_threshold_pct_fall : 30.0 ;
slew_lower_threshold_pct_rise : 30.0 ;
slew_upper_threshold_pct_fall : 70.0 ;
slew_upper_threshold_pct_rise : 70.0 ;
voltage_map (VDD, 1.0);
voltage_map (VDDB, 0.9);
voltage_map (VSS, 0.0);

nom_process : 1.0;
nom_temperature : 25.0;
nom_voltage : 1.1;

operating_conditions(xyz) {
    process : 1.0 ;
    temperature : 25 ;
    voltage : 1.1 ;
    tree_type : "balanced_tree" ;
}
default_operating_conditions : xyz;
... /* Other library-level attributes */

cell (scan_retention_cell) {
    retention_cell : my_scan_ret_cell;
    ... /* Other cell-level attributes and groups */
    pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
    }
    pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
    }
    pg_pin(VDDB) {
        voltage_name : VDDB;
        pg_type : backup_power;
    }
    pin(RET) {
        direction : input;
        capacitance : 0.1;
        related_power_pin : VDDB;
        related_ground_pin : VSS;
        retention_pin(save_restore, "1");
    }
}

```

```

        ... /* Other pin-level attributes and groups */
    }
    pin(D) {
        direction : input;
        capacitance : 0.1;
        related_power_pin : VDD;
        related_ground_pin : VSS;
        ... /* Other pin-level attributes and groups */
    }
    pin(CK) {
        direction : input;
        clock : true;
        capacitance : 0.1;
        related_power_pin : VDD;
        related_ground_pin : VSS;
        ... /* Other pin-level attributes and groups */
    }
    pin(Q) {
        direction : output;
        function : "Q2";
        related_power_pin : VDD;
        related_ground_pin : VSS;
        reference_input : "RET CK q1";
        ... /* Other pin-level attributes and groups */
    }
    pin(q1) {
        direction : internal;
        function : "Q1";
        related_power_pin : VDD;
        related_ground_pin : VSS;
        reference_input : "RET CK SE D SI";
        ... /* Other pin-level attributes and groups */
    }
    retention_condition() {
        power_down_function : "!VDD+VSS";
        required_condition: "!RET";
    }
    latch ("p1 p2,p3,p4,p5", "Q1", "QN1") {
        enable : " p1' + p2' ";
        data_in : " p3'*p4 + p3*p5 ";
        power_down_function : "!VDD+VSS"; /* Latch 1 is powered by Primary
power supply */
    }

    latch ("p1 p2 p3", "Q2", "QN2") {
        enable : " p1 * p2 ";
        data_in : " p3 ";
        power_down_function : "!VDDB+VSS"; /* Latch 2 is powered by Backup
power supply */
    }
    test_cell() {
        pin(SI) {
            direction : input;

```

```

        signal_type : "test_scan_in";
    }
pin(RET) {
    direction : input;
}
pin(D) {
    direction : input;
}
pin(SE) {
    direction : input;
    signal_type : "test_scan_enable";
}
pin(CK) {
    direction : input;
}
latch ("Q1", "QN1") {
    enable : " RET' + CK' ";
    data_in : " D ";
}

latch ("Q2", "QN2") {
    enable : " RET * CK ";
    data_in : " q1 ";
}
pin (q1) {
    direction : internal;
    function : "Q1";
}
pin(Q) {
    direction : output;
    signal_type : "test_scan_out";
    function : "Q2";
} /* End Pin group */
} /* End test_cell group */
} /* End cell group */
} /* End Library group */

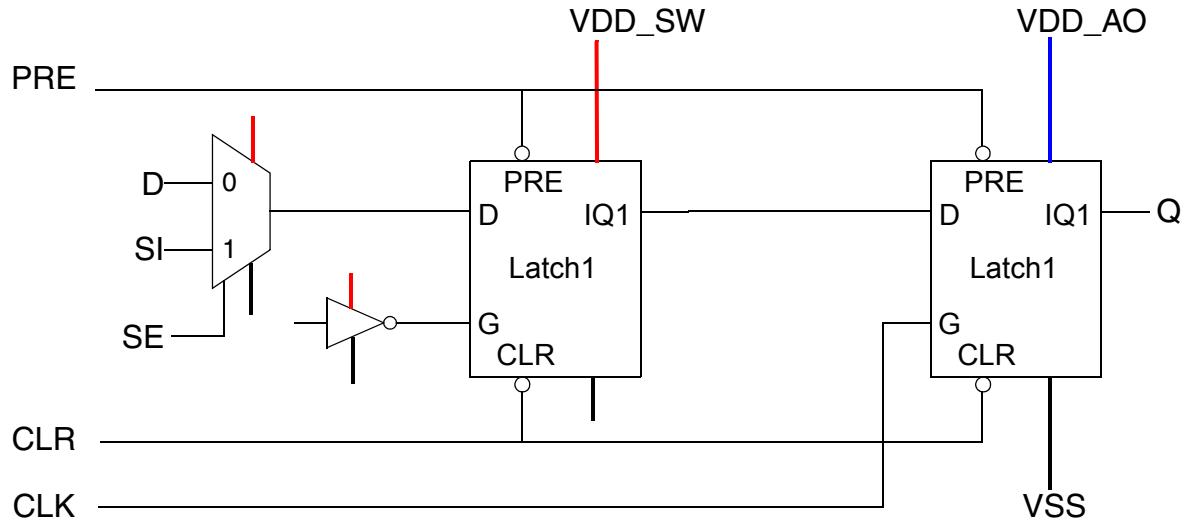
```

Zero-Pin Retention Cell Example

A live-slave retention register has a master-slave structure and two power supplies. The slave latch is a low-leakage block with an always-on supply. The slave latch stores the data during the power down state.

Because this special retention register does not have the save or restore retention control pin, it is also commonly called a zero-pin retention register and its Liberty model must not specify the `retention_pin` attribute on any of the signal pins.

[Figure 13-31](#) shows a zero-pin retention register powered by the always-on supply, VDD_AO, and the switched supply, VDD_SW. The data stored in the slave is maintained while the clock signal, CLK, is in low state and the switched supply is turned off.

Figure 13-31 Zero-Pin Retention Cell

[Example 13-16](#) is a Liberty model of the cell shown in [Figure 13-31](#).

Example 13-16 Zero-Pin Retention Cell Model

```
cell (0_pin_ret_cell) {
    retention_cell : 0_pin_ret;
    pg_pin (VDD_SW) {
        pg_type : primary_power;
        ...
    }
    pg_pin (VDD_AO) {
        pg_type : backup_power;
        ...
    }
    pg_pin (VSS) {
        pg_type : primary_ground;
        ...
    }
    pin (Q) {
        direction : output;
        function : "IQ2";
        related_ground_pin : VSS;
        related_power_pin : VDD_SW;
        power_down_function : "(CLK * !VDD_SW) + !VDD_AO + VSS";
        ...
    }
    pin (D) {
        direction : input;
        related_ground_pin : VSS;
        related_power_pin : VDD_SW;
        ...
    }
    pin (SI) {
```

```

        direction : input;
        related_ground_pin : VSS;
        related_power_pin : VDD_SW;
        ...
    }
pin (SE) {
    direction : input;
    related_ground_pin : VSS;
    related_power_pin : VDD_SW;
    ...
}
pin (CLK) {
    clock : true;
    direction : input;
    related_ground_pin : VSS;
    related_power_pin : VDD_AO;
    ...
}
pin (CLR) {      direction : input;
    related_ground_pin : VSS;
    related_power_pin : VDD_AO;
    ...
}
pin (PRE) {
    direction : input;
    related_ground_pin : VSS;
    related_power_pin : VDD_AO;
    ...
}
latch (IQ1,IQN1) { /* Master latch */
    enable : "!CLK";
    data_in : "!SE * D + SE * SI";
    clear : "!CLR"; /* clear is low active */
    preset : "!PRE"; /* preset is low active */
    clear_preset_var1 : "L";
    clear_preset_var2 : "L";
    power_down_function : "!VDD_SW + VSS";
}
latch (IQ2,IQN2) { /* Slave latch, this is the alive latch*/
    enable : "CLK"; /* clock is high active */
    data_in : "IQ1";
    clear : "!CLR";
    preset : "!PRE";
    clear_preset_var1 : "L";
    clear_preset_var2 : "L";
    power_down_function : "!VDD_AO + VSS";
}
clock_condition() {
    clocked_on : "CLK";
    hold_state : "L";
}
retention_condition() {
    /* Data is stored when clock pin is disabled for alive latch */
}

```

```
/* Data is stored when clear/preset pin is inactive */
/* Pin CLK/CLR/PRE are input pins with backup power */
required_condition : "!CLK * CLR * PRE";
power_down_function : "!VDD_SW + VSS";
}
```

Multibit Retention Scan Cell Examples

The tool supports modeling a multibit retention scan cell that has a serial scan chain with or without a dedicated scan out (SO) pin. To compile this cell model, the tool must recognize the corresponding single-bit retention scan cell and have the multibit-to-single-bit mapping information. For this purpose,

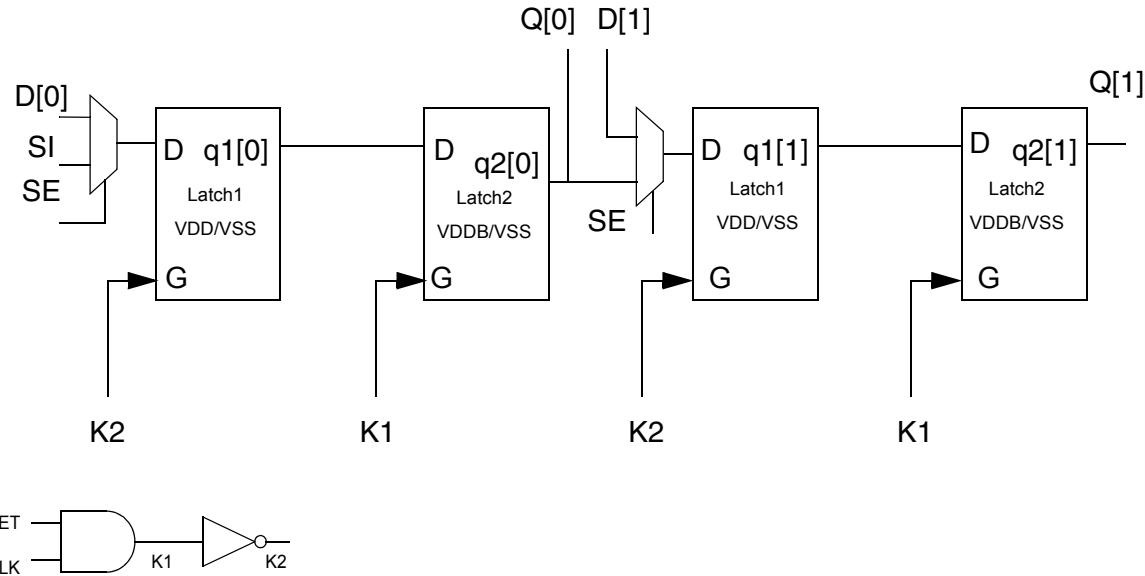
- Model the single-bit retention scan cell using `ff` and `latch` groups.
- Extend this modeling to a multibit cell by applying the `reference_input` attribute on the output and internal pins of the multibit cell.
- Model the nonscan mode of the multibit scan cell using the `ff_bank` and `latch_bank` groups in the `test_cell` group.

Note:

The function of the multibit retention scan cells is natively supported by the Library Compiler `read_lib` command L-2016.06 release onwards. Therefore, you do not need to specify the `single_bit_degenerate` attribute in the multibit retention scan cell models.

For more information about modeling multibit scan cells with serial scan chains, see “[Multibit Scan Cell With Internal Scan Chain](#)” on page 9-14.

Figure 13-32 2-Bit Retention Scan Cell With Master-Slave Latches Without Dedicated Scan Output Pin



[Example 13-17](#) is a Liberty model of the cell shown in [Figure 13-32](#).

Example 13-17 Model of a 2-Bit Retention Scan Cell With Master-Slave Latches and No Dedicated SO Pin

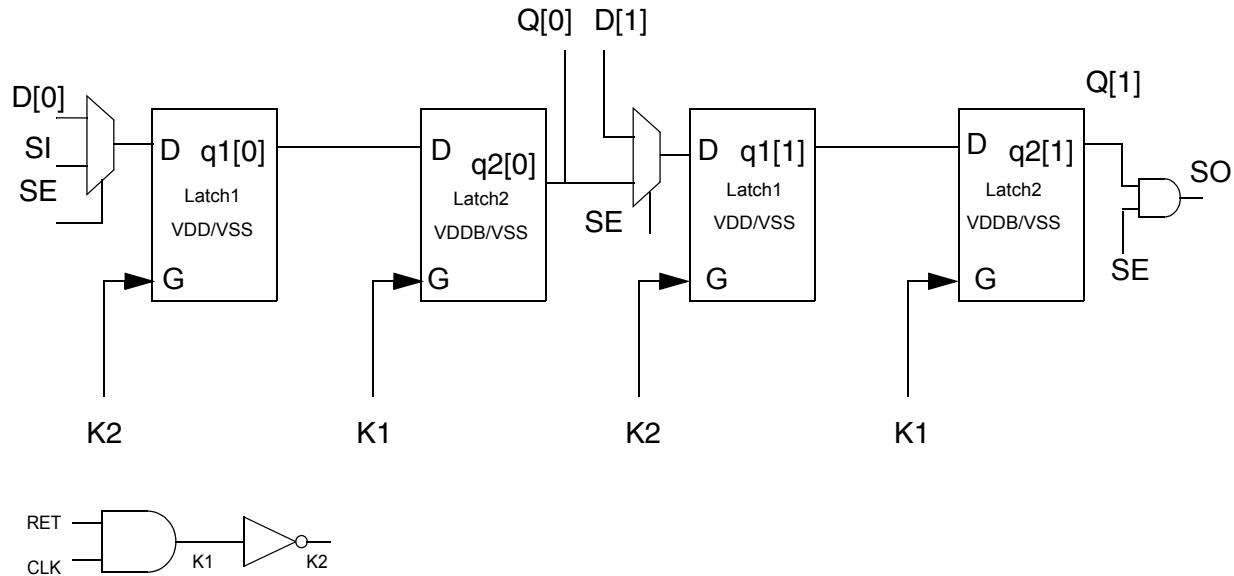
```
cell (MB2_scan_ret_1control_ms) {
    retention_cell : ret_cell;
    pin(CK RET SI SE) {
        ...
    }
    bus(D) {
        ...
    }
    bus(Q) {
        direction : output;
        pin(Q[0]) {
            function : "q2";
            reference_input : "QT[0]";
        }
        pin(Q[1]) {
            function : "q2";
            reference_input : "QT[1]";
        }
    }
    bus(QT) {
        direction : internal;
        pin(QT[0]) {
            function : q1;
            reference_input : "D[0] SI";
        }
    }
}
```

```

    }
    pin(QT[1]) {
        function : q1;
        reference_input : "D[1] Q[0]";
    }
}
latch("pa pb", q1, qn1) {
    enable : "(CK * RET)" ;
    data_in : "pa * !SE + pb * SE" ;
}
latch("pc", q2, qn2) {
    enable : "(CK * RET)" ;
    data_in : "pc" ;
}
test_cell() {
    pin(CK RET SE) {
        ...
    }
    bus(D) { ... }
    pin(SI) {
        signal_type:test_scan_in ;
    }
}
bus(Q) {
    direction : output;
    function : "q2";
    pin(Q[1]) {
        signal_type : test_scan_out ;
    }
}
bus(QT) {
    direction : internal;
    function : q1;
}
latch_bank(q1, qn1, 2) {
    enable : "(CK * RET)" ;
    data_in : "D" ;
}
latch_bank(q2, qn2, 2) {
    enable : "(CK * RET)" ;
    data_in : "QT" ;
}
}
}

```

Figure 13-33 2-Bit Retention Scan Cell With Master-Slave Latches With Dedicated Scan Output (SO) Pin



[Example 13-18](#) is a Liberty model of the cell shown in [Figure 13-33](#).

Example 13-18 Model of a 2-Bit Retention Scan Cell With Master-Slave Latches and Dedicated Scan Out (SO) Pin

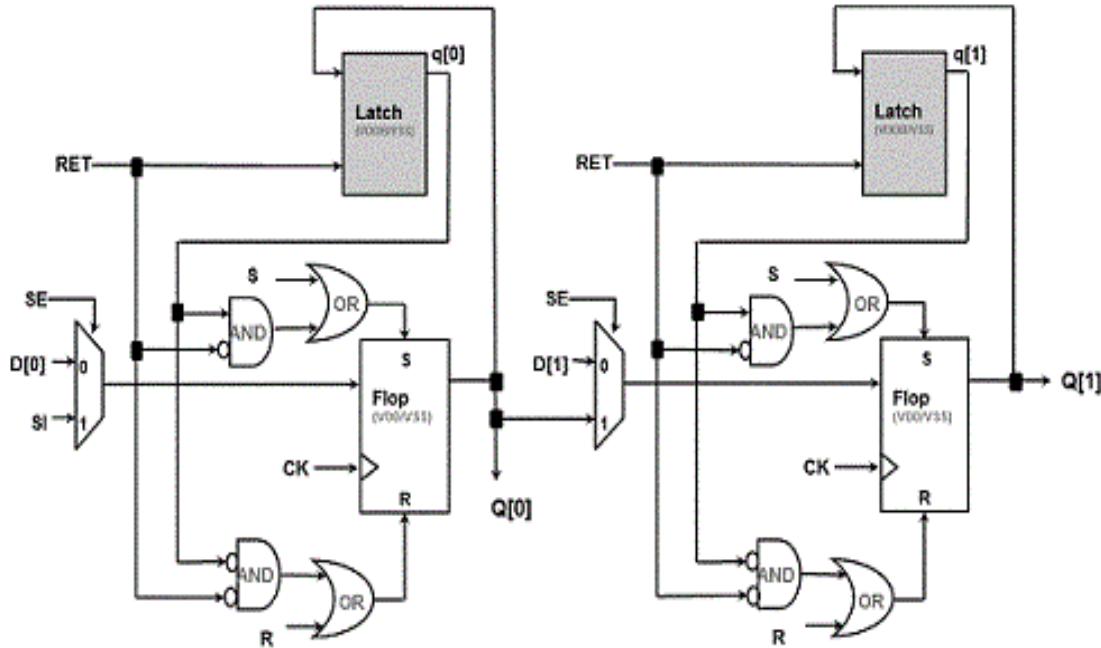
```

cell (multibit_ret) {
    retention_cell : ret_cell;
    pin(CK RET SI SE) {
        ...
    }
    bus(D) {
        ...
    }
    pin(SO) {
        direction : output;
        function : "SE * Q[1]";
    }
    bus(Q) {
        direction : output;
        pin(Q[0]) {
            function : "q2";
            reference_input : "QT[0]";
        }
        pin(Q[1]) {
            function : "q2";
            reference_input : "QT[1]";
        }
    }
    bus(QT) {
        ...
    }
}

```

```
direction : internal;
pin(QT[0]) {
    function : q1;
    reference_input : "D[0] SI";
}
pin(QT[1]) {
    function : q1;
    reference_input : "D[1] Q[0]";
}
latch("pa pb", q1, qn1) {
    enable : "(CK * RET)" ;
    data_in : "pa * !SE + pb * SE" ;
}
latch("pc", q2, qn2) {
    enable : "(CK * RET)" ;
    data_in : "pc" ;
}
test_cell() {
    pin(CK RET SE) {
        ...
    }
    bus(D) { ... }
    pin(SI) {
        signal_type:test_scan_in ;
    }
}
pin(SO) {
    direction : output ;
    signal_type : test_scan_out ;
    test_output_only :"true" ;
}
bus(Q) {
    direction : output;
    function : "q2";
}
bus(QT) {
    direction : internal;
    function : q1;
}
latch_bank(q1, qn1, 2) {
    enable : "(CK * RET)" ;
    data_in : "D" ;
}
latch_bank(q2, qn2, 2) {
    enable : "(CK * RET)" ;
    data_in : "QT" ;
}
}
```

Figure 13-34 2-Bit Retention Scan Cell With Balloon Latches Without Dedicated Scan Output Pin



[Example 13-19](#) is a Liberty model of the cell shown in [Figure 13-34](#).

[Example 13-19 Model of a 2-Bit Retention Scan Cell With Balloon Latches](#)

```
cell(MB2_scan_ret_1cnt_balloon) {
    retention_cell : 1control_ret;
    pin(CK RET SI SE RN SN) {
        ...
    }
    bus(D) {
        ...
    }
    bus(Q) {
        direction : output;
        pin(Q[0]) {
            function : "iq1";
            reference_input : "D[0] SI QT[0]";
        }
        pin(Q[1]) {
            function : "iq1";
            reference_input : "D[1] Q[0] QT[1]";
        }
    }
    bus(QT) {
        direction : internal;
        pin(QT[0]) {
            reference_input : "Q[0]";
        }
    }
}
```

```

        function : iq2;
    }
    pin(QT[1]) {
        reference_input : "Q[1]";
        function : iq2;
    }
}
ff("pa pb pc", iq1,iqn1) {
    clocked_on : "CK";
    next_state : "pa * !SE + pb * SE";
    clear : "(RN + (!RET * !pc))";
    preset : "(SN + (!RET * pc))";
}
latch("pd", iq2,iqn2) {
    enable : "RET";
    data_in : "pd";
    ...
}

test_cell() {
    pin(CK RET SI SE RN SN) {
        ...
    }
    pin(SI) {
        signal_type:test_scan_in
    }
    bus(D) {
        ...
    }
    bus(Q) {
        direction : output;
        function : "iq1";
        pin(Q[1]) {
            signal_type : test_scan_out ;
        }
    }
    bus(QT) {
        direction : internal;
        function : iq2;
    }
}
ff_bank (iq1, iqn1, 2) {
    clocked_on : "CK";
    next_state : "D";
    clear : "(RN + (!RET * !Q_int))";
    preset : "(SN + (!RET * Q_int))";
}
latch_bank (iq2, iqn2, 2) {
    enable : "RET";
    data_in : "Q";
}
}
}
}
```

Retention Cell Library Checks

The Library Compiler tool performs checks for retention cells and issues an error or warning message if certain conditions occur. For a detailed list of library checks for retention cells, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Always-On Cell Modeling

In complex low-power designs, some signals need to be routed through blocks that have been shut down. As a result, a variety of cell categories require “always-on” signal pins. Always-on cells remain powered on by a backup power supply in the region where they are placed even when the main power supply is switched off. The cells also have a secondary backup power pin that supplies the current that is necessary when the main supply is not available.

For tools to recognize always-on cells in the reference library and use them during special always-on synthesis, library models need an attribute that can identify them. The `always_on` attribute identifies always-on cells and pins.

When you run the `read_lib` command, the `always_on` attribute is automatically added to buffer or inverter cells that have input and output pins that are linked to backup power or backup ground PG pins. The following buffers and inverters are automatically identified as always-on cells:

- Cells with one primary power pin, one primary ground pin, and one backup ground pin, if both the input and output signal pins are linked to the primary power and backup ground pins.
- Cells with one primary power pin, one backup power pin, and one primary ground pin, if both the input and output signal pins are linked to the backup power and primary ground pins.
- Cells with one primary power pin, one backup power pin, one primary ground pin, and one backup ground pin if
 - Both the input and output signal pins are linked to the backup power pin and the backup ground pin.
 - Both the input and output signal pins are linked to the primary power pin and the backup ground pin.
 - Both the input and output signal pins are linked to the backup power pin and the primary ground pin.

The `always_on` attribute also identifies always-on input pins so that tools can trace all always-on nets crossing domains that are shut down. Always-on pins are automatically created for the following cells:

- Save and restore pins on retention cells
- Control pins on switch cells
- Enable pins on isolation cells and enable level-shifter cells

The cell library does not require that you specify these cells as always-on cells. If you have other cells that need to be specified as always-on, you can add them to the library cell model.

Always-On Cell Syntax

```
library (library_name) {  
    ...  
    cell (cell_name) {  
        always_on : true;  
        ...  
        pin (pin_name) {  
            always_on : true;  
            ... }  
        ... }  
    ... }
```

always_on Simple Attribute

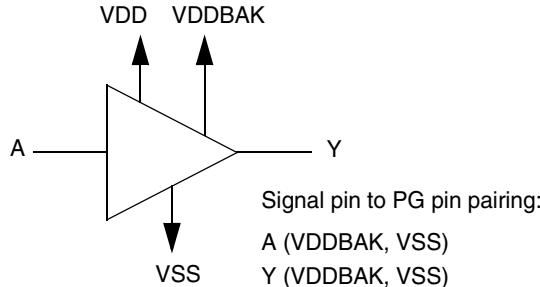
The `always_on` simple attribute models always-on cells and signal pins. During compilation, the `always_on` attribute is automatically added to buffer or inverter cells that have input and output pins that are linked to backup power or backup ground PG pins. The `always_on` attribute is supported at the cell level and at the pin level.

Note:

Some macro cell input pins require that you specify the `always_on` attribute for always-on pins.

Always-On Simple Buffer Example

[Figure 13-35](#) and the example that follows it show a simple always-on cell buffer. In the figure, the A signal pin and the Y signal pin are linked to the VDDBAK and VSS power and ground pin pair.

Figure 13-35 Simple Always-On Cell Buffer

```
library (my_library) {
    ...
    voltage_map (VDD, 1.0);
    voltage_map (VDDBAK, 1.0);
    voltage_map (VSS, 0.0);
    ...

    cell(buffer_type_AO) {
        always_on : true;
        /* The always-on attribute is not required at the cell
           level if the cell is an always-on cell*/
        /* Other cell level information */

        pg_pin(VDD) {
            voltage_name : VDD;
            pg_type : primary_power;
        }

        pg_pin(VDDBAK) {
            voltage_name : VDDBAK;
            pg_type : backup_power;
        }

        pg_pin(VSS) {
            voltage_name : VSS;
            pg_type : primary_ground;
        }
        ...
        pin (A) {
        /* The always-on attribute is not required at the pin
           level if the cell is an always-on cell*/
            related_power_pin : VDDBAK;
            related_ground_pin : VSS;
            /* Other pin level information */
        }
        pin (Y) {
        /* The always-on attribute is not required at the pin
           level if the cell is an always-on cell*/
            function : "A";
        }
    }
}
```

```
related_power_pin : VDDBAK;
related_ground_pin : VSS;
power_down_function : "!VDDBAK + VSS";
/* Other pin level information */
} /* End Pin group */

} /* End Cell group */
...
} /* End Library group */
```

Always-On Cell Library Checks

The Library Compiler tool performs checks for always-on cells and issues an error or warning message if certain conditions occur. For a detailed list of library checks for always-on cells, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

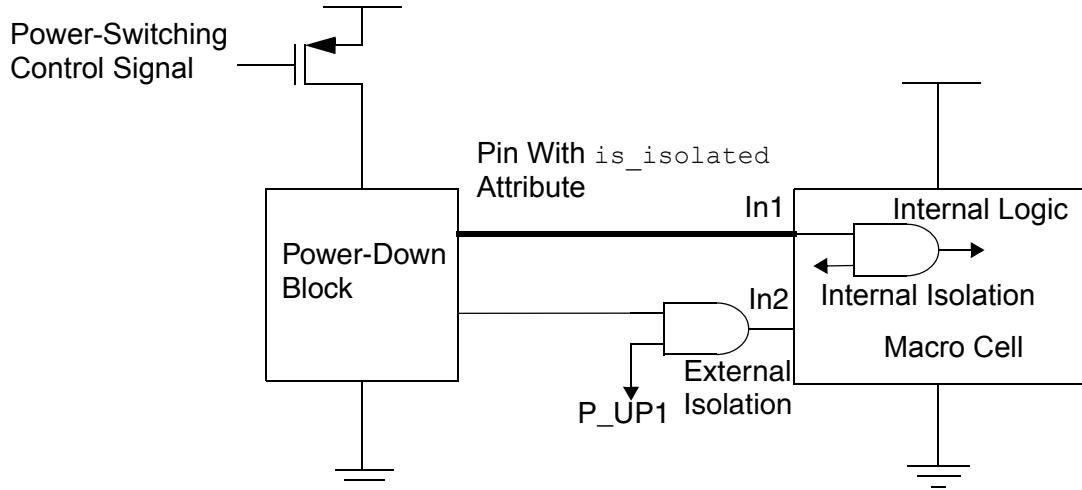
Macro Cell Modeling

Macro cells do not contain internal logic information. At the cell-level, macro cells are modeled using the `is_macro_cell` attribute. To model macro cells for power management, use the power management pin-level attributes described in the following sections.

Macro Cell Isolation Modeling

To indicate that a macro cell is internally isolated and does not require external isolation, set the `is_isolated` attribute. [Figure 13-36](#) shows a macro cell with the `is_isolated` attribute. The macro cell is connected to a power-switching circuit. The macro cell pin, In1 is internally isolated, and In2 is connected to an external isolation cell. When the `is_isolated` attribute is set on the pin, In1, the pin is considered to be internally isolated by the tool.

Figure 13-36 Macro Cell With the `is_isolated` Attribute



[Example 13-20](#) shows the modeling syntax of an internally isolated macro cell.

Example 13-20 Macro Cell Isolation Modeling Syntax

```
cell (cell_name) {
    ...
    is_macro_cell : true;
    pin (pin_name) {
        direction : input | inout | output;
        is_isolated : true | false;
        isolation_enable_condition : Boolean_expression;
        ...
    }
    bus (bus_name) {
        direction : input | inout | output;
        is_isolated : true | false;
        isolation_enable_condition : Boolean_expression;
        ...
    }
    bundle (bundle_name) {
        direction : input | inout | output;
        is_isolated : true | false;
        isolation_enable_condition : Boolean_expression;
        ...
    }
    ...
}
```

[Example 13-21](#) shows a typical model of an internally isolated macro cell.

Example 13-21 Modeling an Internally Isolated Macro Cell by Using the `is_isolated` and `isolation_enable_condition` Attributes

```
library (internal_isolated_pin_example) {
    ...
    type ( bus_type_name ) {
        base_type : array;
        data_type : bit;
        bit_width : 2;
        bit_from : 1;
        bit_to : 0;
    }
    cell ( macro ) {
        is_macro_cell : true;
        pg_pin(GND) {
            voltage_name : VSS;
            pg_type : primary_ground;
        }
        pg_pin(VDDI) {
            voltage_name : VDDH;
            pg_type : primary_power;
        }
        .....
        pin (en) {
            direction : input;
            ...
        }
        pin (in_bus) {
            bus_type : bus_type_name;
            direction : input;
            ...
        }
        pin (sp) {
            direction : input|inout;
            is_isolated : true;
            isolation_enable_condition : "en'";
            related_power_pin : VDDI;
            related_ground_pin : GND;
            .....
        }
        pin (out) {
            direction : output;
            is_isolated : true;
            isolation_enable_condition : "en'";
            related_power_pin : VDDI;
            related_ground_pin : GND;
            .....
        }
        bus (bus_a) {
            bus_type : bus_type_name;
            is_isolated : true;
```

```
isolation_enable_condition : "en' * in_bus * in_bundle";
related_power_pin : VDDI;
related_ground_pin : GND;
...
}
...
}
```

Pin-Level Attributes

This section describes the pin-level attributes of isolated macro cells.

is_isolated Attribute

The `is_isolated` attribute indicates that a pin, bus, or bundle of a macro cell is internally isolated and does not require the insertion of an external isolation cell. The default is `false`.

Note:

The `is_isolated` attribute also supports the internal isolation of pad-cell pins. For more information about modeling internal isolation in I/O pad cells, see “[I/O Pad Cell Internal Isolation Modeling](#)” on page 13-138.

isolation_enable_condition Attribute

The `isolation_enable_condition` attribute specifies the condition of isolation for internally isolated pins, buses, or bundles of a macro cell. When this attribute is defined in a `pin` group, the corresponding Boolean expression can include only input and inout pins. Do not include the output pins of an internally isolated macro cell in the Boolean expression.

When the `isolation_enable_condition` attribute is defined in a `bus` or `bundle` group, the corresponding Boolean expression can include pins, and buses and bundles of the same bit-width. For example, when the Boolean expression includes a bus and a bundle, both of them must have the same bit-width.

All the pins, buses, and bundles specified in the Boolean expression of the `isolation_enable_condition` attribute must have the `always_on` attribute. You do not need to specify the `always_on` attribute in these pin, bus, and bundle groups because the Library Compiler tool automatically infers the `always_on` attribute to be `true` for these pins, buses, and bundles.

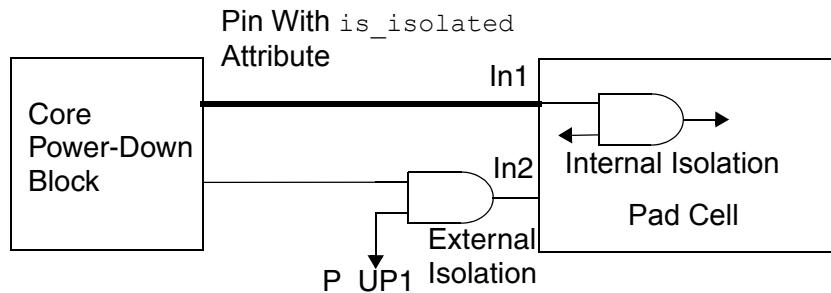
Note:

The `isolation_enable_condition` attribute also supports the internal isolation of pad-cell pins. For more information about modeling internal isolation in I/O pad cells, see “[I/O Pad Cell Internal Isolation Modeling](#)” on page 13-138.

I/O Pad Cell Internal Isolation Modeling

To specify that a pad pin is internally isolated and does not require external isolation, apply the `is_isolated` attribute to the pin. [Figure 13-37](#) shows a pad cell with the `is_isolated` attribute. The pad cell receives the inputs, `In1` and `In2`, from the core. The pad pin, `In1`, is internally isolated, whereas, the pad pin, `In2`, is connected to an external isolation cell. When the `is_isolated` attribute is set on the pin, `In1`, the Library Compiler tool considers the pin to be internally isolated.

Figure 13-37 Pad Cell With the `is_isolated` Attribute



Use the following syntax to model pad cells for internal isolation.

```

cell (cell_name) {
    ...
    pad_cell : true;
    pin (pin_name) {
        direction : input | inout | output;
        is_isolated : true | false;
        isolation_enable_condition : Boolean_expression;
        ...
    }
    bus (bus_name) {
        direction : input | inout | output;
        is_isolated : true | false;
        isolation_enable_condition : Boolean_expression;
        ...
    }
    bundle (bundle_name) {
        direction : input | inout | output;
        is_isolated : true | false;
        isolation_enable_condition : Boolean_expression;
        ...
    }
    ...
}

```

[Example 13-22](#) shows a typical model of an internally isolated pad cell.

Example 13-22 Modeling Pad Cells for Internal Isolation Using the is_isolated and isolation_enable_condition Attributes

```
library (io_with_internal_isolation) {
    ...
    cell (IO_cell) {
        area : 1.0;
        dont_touch : true;
        dont_use : true;
        pad_cell : true;
        pg_pin (VDD) {
            pg_type : primary_power;
            voltage_name : "VDD";
        }
        pg_pin (VDD1) {
            pg_type : primary_power;
            voltage_name : "VDD1";
        }
        pg_pin (VSS) {
            pg_type : primary_ground;
            voltage_name : "VSS";
        }
        pin (PADO) {
            direction : output;
            function : "!PADI";
            three_state : "!TS";
            is_pad : true;
            related_power_pin : VDD1;
            related_ground_pin : VSS;
            timing () {
                related_pin : "TS";
                timing_sense : negative_unate;
                timing_type : three_state_disable;
                ...
            }
            timing () {
                related_pin : "TS";
                timing_sense : positive_unate;
                timing_type : three_state_enable;
                ...
            }
            timing () {
                related_pin : "PADI";
                timing_type : combinational;
                ...
            }
        }
        pin (PO) {
            direction : output;
            function : "!(PADI&TS)";
            related_power_pin : VDD;
        }
    }
}
```

```

related_ground_pin : VSS;
timing () {
    related_pin : "TS";
    timing_type : combinational;
    ...
}
timing () {
    related_pin : "PADI";
    timing_type : combinational;
    ...
}
/* pin ISO is internally isolated */
pin (ISO) {
    is_isolated : true ;
    isolation_enable_condition : !ISO_EN;
    direction : input;
    related_power_pin : VDD1;
    related_ground_pin : VSS;
    capacitance : 0.1;
}
pin (TS) {
    direction : input;
    related_power_pin : VDD;
    related_ground_pin : VSS;
    capacitance : 0.1;
}
pin (PADI) {
    direction : input;
    related_power_pin : VDD1;
    related_ground_pin : VSS;
    capacitance : 0.1;
}
pin (ISO_EN) {
    direction : input;
    related_power_pin : VDD1;
    related_ground_pin : VSS;
    capacitance : 0.1;
}
}
}
}

```

Pin-Level Attributes

This section describes the pin-level attributes of isolated pad cells.

is_isolated Attribute

The `is_isolated` attribute indicates that a pin, bus, or bundle of a pad cell is internally isolated and does not require the insertion of an external isolation cell. The default is `false`.

isolation_enable_condition Attribute

The `isolation_enable_condition` attribute specifies the condition of isolation for internally isolated pins, buses, or bundles of a pad cell. When this attribute is defined in a pin group, the corresponding Boolean expression can include only input and inout pins. Do not include the output pins of a pad cell with internal isolation, in the Boolean expression.

When the `isolation_enable_condition` attribute is defined in a bus or bundle group, the corresponding Boolean expression can include pins, and buses and bundles of the same bit-width. For example, when the Boolean expression includes a bus and a bundle, both of them must have the same bit-width.

All the pins, buses, and bundles specified in the Boolean expression of the `isolation_enable_condition` attribute must have the `always_on` attribute. You do not need to specify the `always_on` attribute in these pin, bus, and bundle groups because the Library Compiler tool automatically infers the `always_on` attribute to be `true` for these pins, buses, and bundles.

I/O Pad Cell Isolation Checks

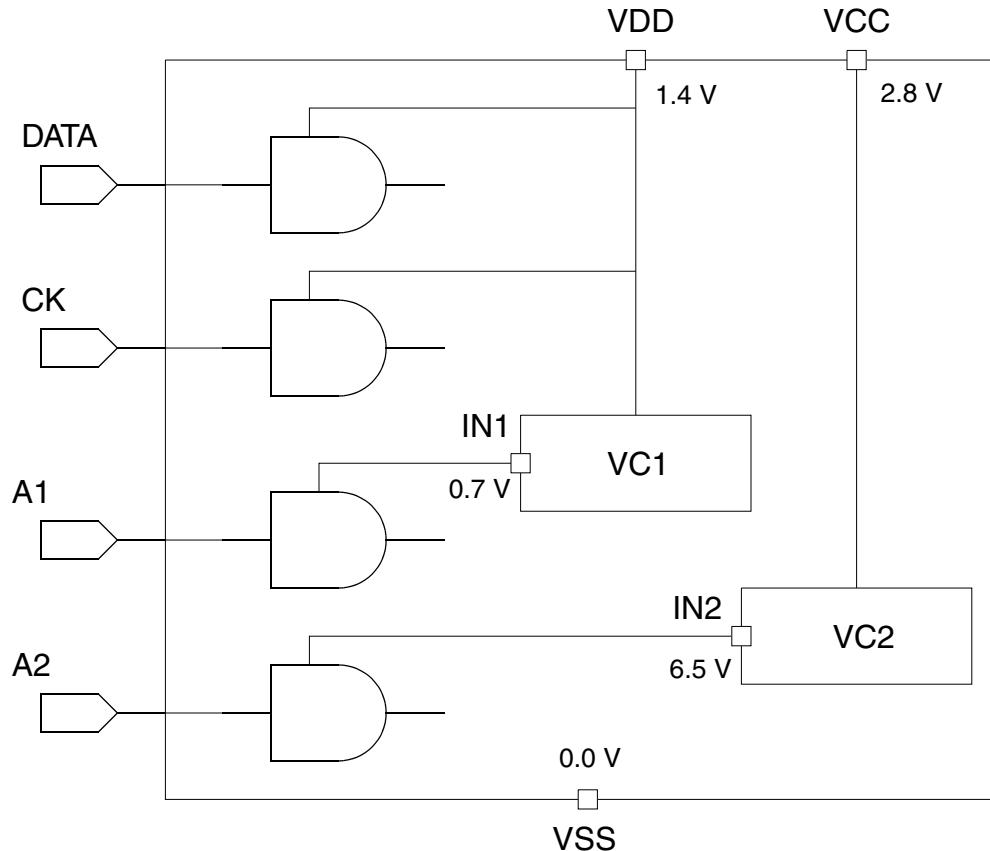
The Library Compiler tool checks for internal isolation conditions of pad cells and issues an error or warning message if certain conditions occur. For a detailed list of these library checks, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Modeling Macro Cells With Internal PG Pins

The Library Compiler tool supports macro cell models with internal PG pins. This capability is useful for modeling macro cells containing internal voltage converters that supply power to the internal subcells.

[Figure 13-38](#) shows the schematic of a macro cell with internal voltage converters, VC1 and VC2. VC1 and VC2 convert the external supply voltages, VDD and VCC, into internal supply voltages, IN1 and IN2, respectively. The pins, IN1 and IN2, are internal PG pins of the macro cell and are used to power the logic connected to the input ports, A1 and A2.

Figure 13-38 Macro Cell With Internal PG Pins



[Example 13-23](#) shows a typical model of the macro cell. The `pg_type` attribute is set to `internal_power`, the `direction` attribute is set to `internal`, and the `pg_function` attribute is set to `VDD` and `VCC`, on the PG pins, `IN1` and `IN2`, respectively.

Example 13-23 Macro Cell Model With Internal PG Pins

```
voltage_map (VDD, 1.4);
voltage_map (VCC, 2.8);
voltage_map (VDD, 1.4);
voltage_map (VCC, 2.8);
voltage_map (VSS, 0.0);
voltage_map (IN1, 0.7);
voltage_map (IN2, 6.5);
cell(new_macro_cell) {
    ...
    pg_pin(VDD) {
        voltage_name : VDD;
        pg_type : primary_power;
    }
}
```

```
pg_pin(VCC) {
    voltage_name : VCC;
    pg_type : primary_power;
}
pg_pin(VSS) {
    voltage_name : VSS;
    pg_type : primary_ground;
}
pg_pin(IN1) {
    voltage_name : IN1;
    pg_type : internal_power;
    direction : internal ;
    pg_function : VDD;
}
pg_pin(IN2) {
    voltage_name : IN2;
    pg_type : internal_power;
    direction : internal ;
    pg_function : VCC;
}
...
pin(CK) {
    related_power_pin : VDD ;
    related_ground_pin : VSS ;
    direction : input ;
    ...
}
pin(A1) {
    related_power_pin : IN1 ;
    related_ground_pin : VSS ;
    direction : input ;
    ...
}
pin(A2) {
    related_power_pin : IN2 ;
    related_ground_pin : VSS ;
    direction : input ;
}
...
}
```

Note:

You do not need to specify the `switch_cell_type` and `switch_function` attributes for this macro cell because it does not include any built-in switches.

Low Power Macro Cell Modeling

Complex macro cell designs use multiple power management techniques simultaneously. These include internal voltage shift circuitry, power-gating logic, and internal clamp logic at the output. To model low-power complex macro cells, you can use the following attributes:

- Partial power down
 - “[permit_power_down Attribute](#)” on page 13-62
 - “[alive_during_partial_power_down Attribute](#)” on page 13-62
- Isolation
 - “[isolation_cell_data_pin Attribute](#)” on page 13-61
 - “[isolation_cell_enable_pin Attribute](#)” on page 13-61
 - “[clock_isolation_cell_clock_pin Attribute](#)” on page 13-69
- Level-shifting
 - “[level_shifter_data_pin Attribute](#)” on page 13-37
 - “[level_shifter_enable_pin Attribute](#)” on page 13-37
- Clamp outputs
 - “[clamp_0_function Attribute](#)” on page 13-39
 - “[clamp_1_function Attribute](#)” on page 13-39
 - “[clamp_z_function Attribute](#)” on page 13-39
 - “[clamp_latch_function Attribute](#)” on page 13-40
 - “[illegal_clamp_condition Attribute](#)” on page 13-40
- Specify voltage range
 - “[input_voltage_range Attribute](#)” on page 13-36
 - “[output_voltage_range Attribute](#)” on page 13-36

[Example 13-24](#) shows the Liberty model of a typical low power macro cell. The internal supply voltage, VDD2, is different from the supply voltage, VDD1. The cell has a large input voltage range from 0.7V to 1.25V, and can be operated at different voltages. When VDD2 is switched off and the cell is isolated, the cell output clamps to 0.

Example 13-24 A Low-Power Complex Macro Cell

```
library (iso_macro) {
    voltage_map (VDD1, 1.8);
```

```
voltage_map (VDD2, 1.0);
voltage_map (VDD, 1.05);
voltage_map (VDD_AON, 0.9);
voltage_map (VSS, 0.0);
cell(my_macro) {
    is_macro_cell : true;
    switch_cell_type : fine_grain;
    pg_pin(VDD1) {
        pg_type : primary_power;
        voltage_name : VDD1;
        direction : input;
    }
    pg_pin(VDD2) {
        direction : internal;
        voltage_name : VDD2;
        pg_type : internal_power;
        switch_function : "!pwr_on";
        pg_function : "VDD1";
        permit_power_down : true;
    }
    pg_pin(VSS) {
        voltage_name : VSS;
        pg_type : primary_ground;
        direction : input;
    }
    pin(pwr_on) {
        direction : input;
        switch_pin : true;
        related_power_pin : VDD1;
        related_ground_pin : VSS;
        input_signal_level : VDD_AON;
        input_voltage_range (0.7, 1.25);
    }
    /* other input pins */
    pin(iso_in) {
        direction : input;
        level_shifter_data_pin : true;
        related_power_pin : VDD2;
        related_ground_pin : VSS;
        input_signal_level: VDD_AON;
        input_voltage_range (0.7, 1.25);
    }
    pin(iso_out) {
        direction : input;
        isolation_cell_enable_pin : true;
        related_power_pin : VDD2;
        related_ground_pin : VSS;
        input_signal_level: VDD_AON;
        input_voltage_range (0.7, 1.25);
        alive_during_partial_power_down : true;
    }
    pin(out) {
        direction : output;
```

```

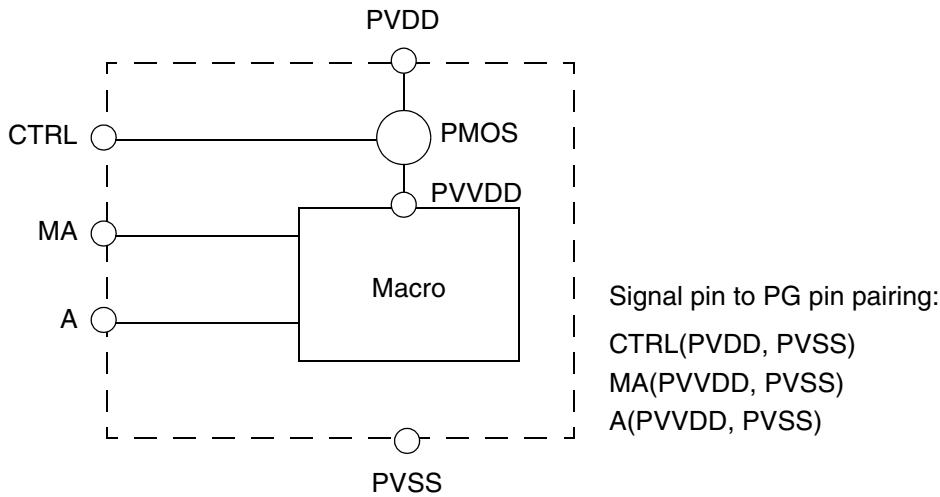
is_isolated : true;
isolation_enable_condition : iso_out;
related_power_pin : VDD2;
related_ground_pin : VSS;
power_down_function : "(!VDD2 * !iso_out) + VSS";
alive_during_partial_power_down : true;
clamp_0_function : "iso_out";
...
}
} /* end cell */
} /* end library */

```

Macro Cell With Fine-Grained Internal Power Switches

[Figure 13-39](#) and the example that follows it show a macro cell with fine-grained internal power switches. In the figure, the CTRL signal pin is linked to the PVDD and PVSS power and ground pin pair, and the MA signal pin and the A signal pin are linked to the PVVDD and PVSS power and ground pin pair.

Figure 13-39 Macro Cell With Fine-Grained Power Switch Schematics



```

library (macro_switch) {
...
Voltage_map (PVDD, 1.0);
Voltage_map (PVVDD, 1.0);
Voltage_map (PVSS, 0.0);

operating_conditions(XYZ) {
    process : 1.0;
    voltage : 1.0;
    temperature : 25.0;
}

```

```
default_operating_conditions : XYZ;

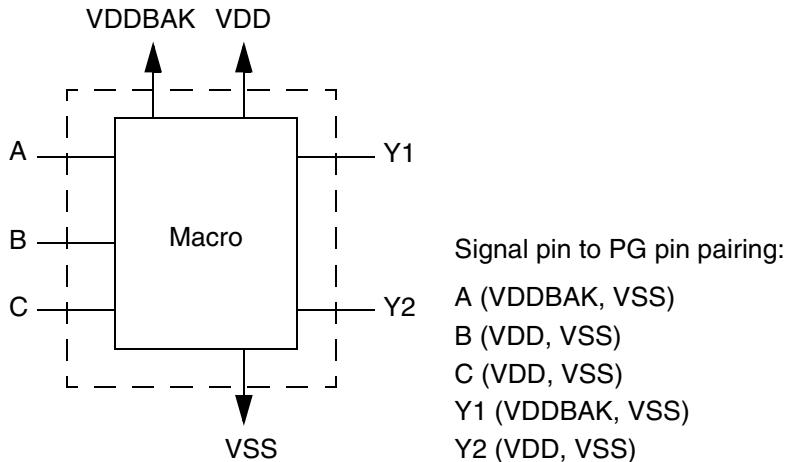
cell(MACRO) {
    is_macro_cell : true;
    switch_cell_type : fine_grain;
    ...

    pg_pin(PVDD) {
        voltage_name : PVDD;
        pg_type : primary_power;
        direction : input;
    }
    pg_pin(PVSS) {
        voltage_name : PVSS;
        pg_type : primary_ground;
        direction : input;
    }
    pg_pin(PVVDD) {
        voltage_name : PVVDD;
        pg_type : internal_power;
        direction : internal;
        switch_function : "CTRL";
        pg_function : "PVDD";
    }
    pin(CTRL) {
        direction : input;
        switch_pin : true;
        related_power_pin : PVDD;
        related_ground_pin : PVSS;
        ...
    }
    pin(A) {
        direction : input;
        related_power_pin : PVVDD;
        related_ground_pin : PVSS;
        ...
    }
    pin(MA) {
        direction : input;
        power_down_function : "!PVDD + PVSS";
        related_power_pin : PVVDD;
        related_ground_pin : PVSS;
        ...
    }
}
```

Macro Cell With an Always-On Pin Example

[Figure 13-40](#) and the example that follows it show a macro cell with one always-on pin. In the figure, the A signal pin and Y1 signal pin are linked to the VDDBAK and VSS power and ground pin pair. The B, C, and Y2 signal pins are linked to the VDD and VSS power and ground pin pair.

Figure 13-40 Macro Cell With an Always-On Pin



```

library (my_library) {
  ...
  voltage_map (VDD, 2.0) ;
  voltage_map (VSS, 0.1) ;
  voltage_map (VDDBAK, 1.0) ;
  ...
  cell(Macro_cell_with_AO_pins) {
    /* other cell level information */

    pg_pin(VDD) {
      voltage_name : VDD;
      pg_type : primary_power;
    }
    pg_pin(VSS) {
      voltage_name : VSS;
      pg_type : primary_ground;
    }
    pg_pin(VDDBAK) {
      voltage_name : VDDBAK;
      pg_type : backup_power;
    }
    ...
    pin (A) {
      always_on : true;
    }
  }
}

```

```
    related_power_pin : VDDBAK;
    related_ground_pin : VSS;
/* Other pin level information */
}
pin (B C) {
    related_power_pin : VDD;
    related_ground_pin : VSS;
/* Other pin level information */
}
pin (Y1) {
    always_on : true;
    function : "A";
    related_power_pin : VDDBAK;
    related_ground_pin : VSS;
    power_down_function : "!VDD + !VDDBAK + VSS";
/* Other pin specific information */
} /* End Pin group */
pin (Y2) {
    function : "A";
    related_power_pin : VDD;
    related_ground_pin : VSS;
    power_down_function : "!VDD + !VDDBAK + VSS";
/* Other pin specific information */
} /* End Pin group */
...
} /* End Cell group */
...
} /* End Library group */
```

Macro Cell Isolation Checks

The Library Compiler tool checks for conditions of macro cells and issues an error or warning message if certain conditions occur. For a detailed list of these library checks, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

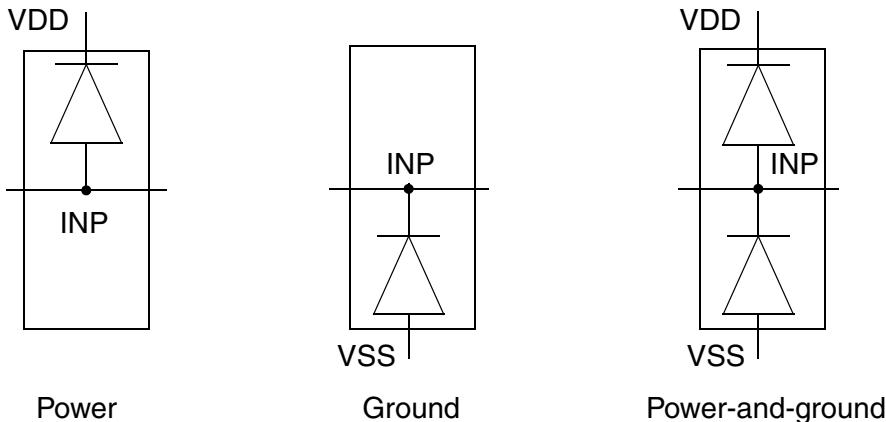
Modeling Antenna Diodes

Modeling antenna diodes includes modeling antenna-diode cells and cells with built-in antenna-diode ports.

Antenna-Diode Cell Modeling

An antenna-diode cell has only one input to a diode that discharges electrical charges. The cell is typically inserted at the boundary between two power domains and can be placed in any one of the power domains. [Figure 13-41](#) shows the three types of antenna-diode cells.

Figure 13-41 Types of Antenna-Diode Cells



In multivoltage designs, the power type antenna-diode cell is connected to VDD of a power domain where VSS is shut down. The ground type antenna-diode cell is connected to VSS of a power domain where VDD is shut down. The power-and-ground type antenna-diode cell is connected to both VDD and VSS. To eliminate leakage paths that can result in chip failure, the correct type of antenna-diode cell must be inserted.

The modeling syntax of the antenna-diode cell is as follows:

```
cell (cell name) {
    antenna_diode_type : power | ground | power_and_ground;
    pin (pin name) {
        antenna_diode_related_power_pins : power pin name;
        antenna_diode_related_ground_pins : ground pin name;
        ...
    }
    ...
}
```

In a library, a cell that has a single pin with the direction attribute set to `input` or `inout` is considered to be an antenna-diode cell.

Cell-Level Attribute

antenna_diode_type Attribute

The `antenna_diode_type` attribute specifies the type of antenna-diode cell. Valid values are `power`, `ground`, and `power_and_ground`.

Pin-Level Attributes

antenna_diode_related_power_pins Attribute

The `antenna_diode_related_power_pins` attribute specifies the related power pin of the antenna-diode cell. Apply the `antenna_diode_related_power_pins` attribute to the input pin of the cell.

antenna_diode_related_ground_pins Attribute

The `antenna_diode_related_ground_pins` attribute specifies the related ground pin of the antenna-diode cell. Apply the `antenna_diode_related_ground_pins` attribute to the input pin of the cell.

Antenna-Diode Cell Modeling Example

[Example 13-25](#) shows a typical model of the antenna-diode cell.

Example 13-25 Antenna-Diode Cell Model

```
library (antenna_library)
...
cell (power_diode_cell) {
    antenna_diode_type : "power";
    pg_pin (VDD) {
        voltage_name : "VDD";
        pg_type : "primary_power";
    }
    pin (INP) {
        antenna_diode_related_power_pins : "VDD";
        direction : "input";
    }
}
cell (ground_diode_cell) {
    antenna_diode_type : "ground";
    pg_pin (VSS) {
        voltage_name : "VSS";
        pg_type : "primary_ground";
    }
    pin (INP) {
        antenna_diode_related_ground_pins : "VSS";
        direction : "input";
    }
}
cell (pg_diode_cell) {
    antenna_diode_type : "power_and_ground";
    pg_pin (VDD) {
        voltage_name : "VDD";
        pg_type : "primary_power";
    }
}
```

```

    pg_pin (VSS) {
        voltage_name : "VSS";
        pg_type : "primary_ground";
    }
    pin (INP) {
        antenna_diode_related_power_pins : "VDD";
        antenna_diode_related_ground_pins : "VSS";
        direction : "input";
    }
}
}

```

Modeling Cells With Built-In Antenna-Diode Ports

To model a cell with a built-in antenna-diode pin, specify the `antenna_diode_type` attribute on the pin.

The modeling syntax of the cell with a built-in antenna-diode pin is as follows:

```

cell (cell name) {
    ...
    pin (pin name) {
        antenna_diode_type : power | ground | power_and_ground;
        antenna_diode_related_power_pins : "power_pin1 power_pin2";
        antenna_diode_related_ground_pins : "ground_pin1 ground_pin2";
        ...
    }
    pin (pin name) {
        ...
    }
    ...
}

```

Pin-Level Attributes

antenna_diode_type Attribute

The `antenna_diode_type` attribute specifies the type of antenna-diode pin. Valid values are `power`, `ground`, and `power_and_ground`.

antenna_diode_related_power_pins Attribute

The `antenna_diode_related_power_pins` attribute specifies the related power pins for the antenna-diode pin. Apply the `antenna_diode_related_power_pins` attribute to the antenna-diode pin.

antenna_diode_related_ground_pins Attribute

The `antenna_diode_related_ground_pins` attribute specifies the related ground pins for the antenna-diode pin. Apply the `antenna_diode_related_ground_pins` attribute to the antenna-diode pin.

Antenna-Diode Pin Modeling Example

[Example 13-26](#) shows a typical model of a cell with a built-in antenna-diode pin.

Example 13-26 A Cell Model With Built-In power_and_ground Antenna-Diode Pin Port

```
cell (cell_with_internal_diode_port)
  area : "1.0";
  pg_pin (VDD) {
    voltage_name : "VDD";
    pg_type : "primary_power";
  }
  pg_pin (VDD1) {
    voltage_name : "VDD1";
    pg_type : "primary_power";
  }
  pg_pin (VSS) {
    voltage_name : "VSS";
    pg_type : "primary_ground";
  }
  pg_pin (VSS1) {
    voltage_name : "VSS1";
    pg_type : "primary_ground";
  }
  pin (antenna_diode) {
    antenna_diode_type: power_and_ground;
    antenna_diode_related_power_pins : "VDD VDD1";
    antenna_diode_related_power_pins : "VSS VSS1";
    direction : "input";
    capacitance : "1.0";
  }
  pin (INP1) {
    direction : "input";
    capacitance : "1.0";
  }
  pin (INP2) {
    direction : "input";
    capacitance : "1.0";
  }
  pin (OUT1) {
    related_power_pin : "VDD";
    related_ground_pin : "VSS";
    direction : "output";
  }
  pin (OUT1) {
    related_power_pin : "VDD";
```

```
    related_ground_pin : "VSS";
    direction : "output";
}
}
```

Antenna-Diode Cell Checks

The Library Compiler tool checks the conditions of cells and pins with the `antenna_diode_type` attribute and issues errors or warning messages if certain conditions occur. For a detailed list of these library checks, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter of the *Library Quality Assurance System User Guide*.

14

Composite Current Source Modeling

This chapter provides an overview of composite current source modeling (CCS). It covers the syntax for CCS modeling in the following sections:

- [Modeling Cells With Composite Current Source Information](#)
- [Representing Composite Current Source Driver Information](#)
- [Representing Composite Current Source Receiver Information](#)
- [Two-Segment Receiver Capacitance Model](#)
- [Multisegment Receiver Capacitance Model](#)
- [CCS Retain Arc Support](#)
- [Composite Current Source Driver and Receiver Model Example](#)
- [Checking CCS Library Models](#)

Modeling Cells With Composite Current Source Information

Composite current source (CCS) models include driver and receiver models. A driver model can be used with or without the receiver model.

Nanometer transistor geometries with nonlinear waveforms, nonlinear interconnect delays, and high circuit speeds require accurate delay calculation. CCS models support driver complexity by using a time- and voltage- dependent current source with an infinite drive resistance. The driver model maps the arbitrary transistor behavior for lumped loads to that for an arbitrary detailed parasitic network, which results in high accuracy.

In the receiver model, the input capacitance of a receiver is dynamically adjusted during the voltage transitions.

Representing Composite Current Source Driver Information

Specify the nonlinear delay information based on input slew and output load at the pin level by defining a current lookup table in a timing group.

Composite Current Source Lookup Tables

To define the lookup tables, use the following groups and attributes:

- `output_current_template` group in the `library` group
- `output_current_rise` and `output_current_fall` groups in the `timing` group

Defining the `output_current_template` Group

Use this library-level group to create templates of common information that multiple current vectors can use. A table template specifies the composite current source driver model and the breakpoints for the axis. Specifying `index_1`, `index_2`, and `index_3` values at the library level is optional.

`output_current_template` Syntax

```
library(name_id) {  
    ...  
    output_current_template(template_name_id) {  
        variable_1: input_net_transition;  
        variable_2: total_output_net_capacitance;  
        variable_3: time;  
        ...  
    }  
}
```

```

    }
...
}

```

Template Variables for CCS Driver Models

The table template specifying composite current source driver models can have three variables: variable_1, variable_2, and variable_3. The valid values for variable_1 and variable_2 are input_net_transition and total_output_net_capacitance. The only valid value for variable_3 is time.

output_current_template Example

```

library (new_lib) {
...
  output_current_template (CCT) {
    variable_1: input_net_transition;
    variable_2: total_output_net_capacitance;
    variable_3: time;
    ...
  }
...
}

```

Defining the Lookup Table Output Current Groups

To specify the output current for the nonlinear table model, use the `output_current_rise` and `output_current_fall` groups within the `timing` group.

output_current_rise Syntax

```

timing() {
  output_current_rise () {
    vector (template_name_id) {
      reference_time : float;
      index_1 (float);
      index_2 (float);
      index_3 ("float,..., float");
      values("float,..., float");
    }
  }
}

```

vector Group

Define the `vector` group in the `output_current_rise` or `output_current_fall` group. This group stores current information for a particular input slew and output load.

reference_time Simple Attribute

Define the `reference_time` simple attribute in the `vector` group. The `reference_time` attribute represents the time at which the input waveform crosses the rising or falling input delay threshold.

Template Variables for CCS Driver Models

The table template specifying composite current source driver models can have three variables: `variable_1`, `variable_2`, and `variable_3`. The valid values for `variable_1` and `variable_2` are `input_net_transition` and `total_output_net_capacitance`. The only valid value for `variable_3` is `time`.

The index value for `input_net_transition` or `total_output_net_capacitance` is a single floating-point number. The index values for `time` are a list of floating-point numbers.

The `values` attribute defines a list of floating-point numbers that represent the current values of the driver model.

vector Group Example

```
library (new_lib) {
    ...
    output_current_template (CCT) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        variable_3: time;
    }
    ...
    timing() {
        output_current_rise() {
            vector(CCT) {
                reference_time : 0.05;
                index_1(0.1);
                index_2(2.1);
                index_3("1.0, 1.5, 2.0, 2.5, 3.0");
                values("1.1, 1.2, 1.4, 1.3, 1.5");
                ...
            }
        }
        output_current_fall() {
            vector(CCT) {
                reference_time : 0.05;
                index_1(0.1);
                index_2(2.1);
                index_3("1.0, 1.5, 2.0, 2.5, 3.0");
                values("1.1, 1.2, 1.4, 1.3, 1.5");
                ...
            }
        }
    }
}
```

```
    }  
}
```

Checking CCS Driver Models

The Library Compiler tool checks CCS driver models and reports any problems it encounters. For information about the checks the Library Compiler tool performs for CCS driver models, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Representing Composite Current Source Receiver Information

Switching transistors have nonlinear capacitance. At an input or inout node, the collective capacitance of all the transistors connected to that node is modeled as the receiver capacitance. The Miller effect due to the output switching in the opposite direction adds to the nonlinearity. The capacitances are dependent on the input slew and output load.

CCS receiver models are of two types:

- Two-segment receiver capacitance model

The voltage rise or fall at an input or inout node is divided into two segments and the corresponding capacitance values are stored as `receiver_capacitance1` and `receiver_capacitance2` lookup tables.

- Multisegment capacitance model

The voltage rise or fall at an input or inout node is divided into multiple segments and the corresponding capacitance values are stored as lookup tables. The number of segments is represented by the letter N. This model better represents the nonlinearity of the net receiver capacitance.

For the tool to compile the multisegment receiver capacitance model, set the following variable to `true` before running the `read_lib` command:

```
lc_shell> set lc_enhanced_ccs true
```

Comparison Between Two-Segment and Multisegment Receiver Models

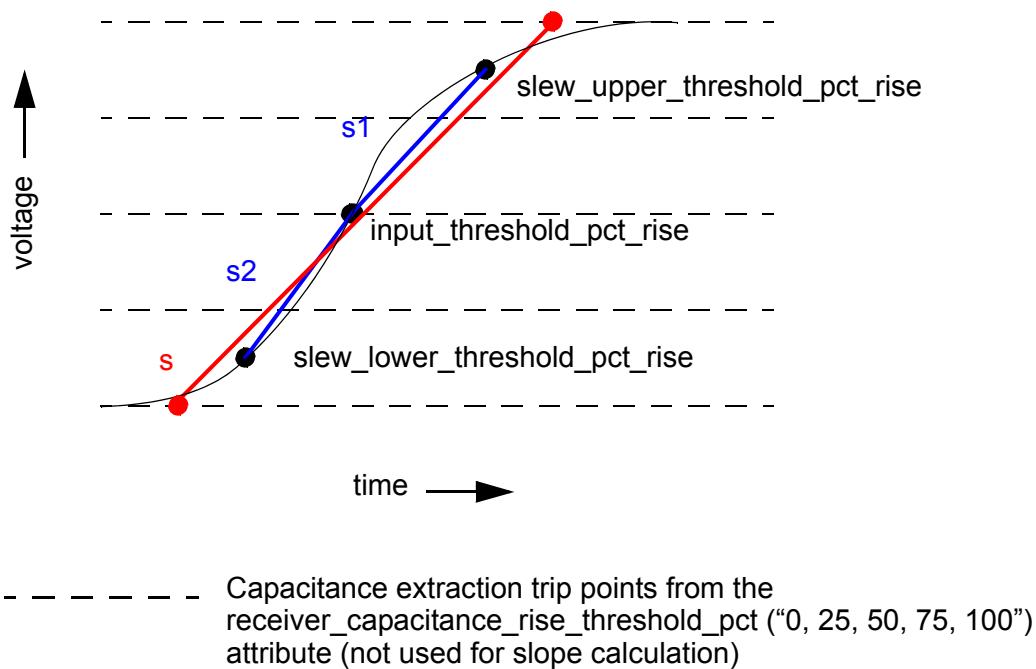
Downstream tools might achieve different results with the two-segment and the multisegment receiver capacitance models.

This is because the number of segments are different. Also, the multisegment model uses the input slew value of the library-level `slew_derate_from_library` attribute, shown with

the solid red line in [Figure 14-1](#). This slew is used for the `input_net_transition(index_1)` values of all the N segments, regardless of the local waveform slopes.

The two-segment model uses the local slope of the two segments (s_1 and s_2 in [Figure 14-1](#)), shown with the solid blue lines. So even at $N = 2$, the multisegment and two-segment model values might not match.

Figure 14-1 Slope Calculation in Multisegment and Two-Segment Receiver Models



Two-Segment Receiver Capacitance Model

Define the CCS receiver model

- At the pin level using the `receiver_capacitance` group.

Specify the `receiver_capacitance1_rise`, `receiver_capacitance1_fall`, `receiver_capacitance2_rise`, `receiver_capacitance2_fall` groups in the `receiver_capacitance` group.

The pin-level definition is not a function of the output capacitance and is useful when there are no forward timing arcs.

- At the timing level by specifying the `receiver_capacitance1_rise`, `receiver_capacitance1_fall`, `receiver_capacitance2_rise`, `receiver_capacitance2_fall` groups in the timing group.

Syntax

The following is the Liberty syntax of CCS timing receiver models.

```

cell(cell_name) {
    mode_definition (mode_name) {
        mode_value(name_string) {
            when : "Boolean_expression";
            sdf_cond : "Boolean_expression";
        } ...
    } ...
    pin(pin_name) {
        direction : input; /* or "inout" */
        receiver_capacitance() {
            when : "Boolean_expression";
            char_when : "Boolean_expression";
            mode (mode_name, mode_value);
            receiver_capacitance1_rise (lu_template_name) {
                index_1("float, ..., float");
                index_2("float, ..., float");
                values("float, ..., float");
            }
            receiver_capacitance1_fall (lu_template_name) { ... }
            receiver_capacitance2_rise (lu_template_name) { ... }
            receiver_capacitance2_fall (lu_template_name) { ... }
        }
    }
    pin(pin_name) {
        direction : output; /* or "inout" */
        timing() {
            when : "Boolean_expression";
            mode (mode_name, mode_value);
            ...
            receiver_capacitance() {
                receiver_capacitance1_rise (lu_template_name) {
                    index_1("float, ..., float");
                    index_2("float, ..., float");
                    values("float, ..., float");
                }
                receiver_capacitance1_fall (lu_template_name) { ... }
                receiver_capacitance2_rise (lu_template_name) { ... }
                receiver_capacitance2_fall (lu_template_name) { ... }
            }
        }
    } ...
}

```

Defining the receiver_capacitance Group at the Pin Level

To model the CCS receiver capacitance at the pin level, define the `receiver_capacitance` group in the `pin` group. In the `receiver_capacitance` group, specify the `receiver_capacitance1_rise`, `receiver_capacitance1_fall`, `receiver_capacitance2_rise`, and `receiver_capacitance2_fall` groups. Each of these groups specify a lookup table. The lookup table can be one-dimensional, two-dimensional, or scalar.

The index values in the `index_1` and `index_1` attributes are a list of ascending floating-point numbers. The `values` attribute defines a list of floating-point numbers that represent the capacitances of the receiver model.

Define the template of the lookup table in the library-level `lu_table_template` group. To use the template, specify the name of the `lu_table_template` group as the receiver capacitance group name.

Defining the lu_table_template Group

The `lu_table_template` group defines the template for the following groups specified in the `receiver_capacitance` group:

- `receiver_capacitance1_rise`
- `receiver_capacitance1_fall`
- `receiver_capacitance2_rise`
- `receiver_capacitance2_fall`

The template definition includes the `variable_1`, `variable_2`, `index_1`, and `index_2` attributes. The `variable_1` and `variable_2` attributes specify the index variables of the lookup tables of the receiver capacitance groups. The `index_1` and `index_2` attributes specify the numerical values of these variables.

The values of the `variable_1` and `variable_2` attributes are `input_net_transition` and `total_output_net_capacitance`.

You can specify the following types of lookup tables in receiver capacitance groups:

- One-dimensional tables with a single index, such as `input_net_transition`
- Two-dimensional tables with the indexes, `input_net_transition` and `total_output_net_capacitance`
- A scalar that has a single variation value irrespective of the `input_net_transition` and `total_output_net_capacitance` values

lu_table_template Group Syntax

```
library(name) {
  ...
  lu_table_template(template_name) {
    variable_1: input_net_transition;
    variable_2: total_output_net_capacitance;
    index_1 ("float,..., float");
    index_2 ("float,..., float");
  ...
}
...
}
```

Conditional Data Modeling

Use the `mode`, `when`, and `char_when` attributes in the `receiver_capacitance` group for conditional data modeling in CCS receiver models.

when Attribute

The `when` attribute specifies the Boolean condition for the input state of the receiver capacitance timing arc.

char_when Attribute

The `char_when` attribute specifies the input state at which the receiver capacitance timing arc was characterized. Correlation tools can use this input state for SPICE validation.

mode Attribute

The complex `mode` attribute specifies the current mode of the cell. When defined in the `receiver_capacitance` group, the `mode` attribute specifies the receiver capacitance for the given mode. If you specify the `mode` attribute, you must define the `mode_definition` group at the cell level.

Examples

[Example 14-1](#) shows a receiver capacitance model with conditional data, that is, the `when` and `mode` attributes.

Example 14-1 Modeling Receiver Capacitance With when and mode Attributes

```
library(new_lib) {
  ...
  output_current_template(CCT) {
    variable_1: input_net_transition;
    variable_2: total_output_net_capacitance;
    variable_3: time;
    index_1("0.1, 0.2");
    index_2("1, 2");
```

```

    index_3("1, 2, 3, 4, 5");
}
lu_table_template(LTT1) {
    variable_1: input_net_transition;
    index_1("0.1, 0.2, 0.3, 0.4");
}
lu_table_template(LTT2) {
    variable_1: input_net_transition;
    variable_2: total_output_net_capacitance;
    index_1("0.1, 0.2");
    index_2("1, 2");
}
...
cell(my_cell) {
    ...
    mode_definition(rw) {
        mode_value(read) {
            when : "I";
            sdf_cond : "I == 1";
        }
        mode_value(write) {
            when : "!I";
            sdf_cond : "I == 0";
        }
    }
    pin(I) /* pin-based receiver model defined for pin 'A' */
    direction : input;
    /* receiver capacitance for condition 1 */
    receiver_capacitance() {
        when : "I"; /* or using mode as next commented line */
        /* mode (rw, read); */
        receiver_capacitance1_rise(LTT1) {
            values("1, 2, 3, 4");
        }
        receiver_capacitance1_fall(LTT1) {
            values("1, 2, 3, 4");
        }
        receiver_capacitance2_rise(LTT1) {
            values("1, 2, 3, 4");
        }
        receiver_capacitance2_fall(LTT1) {
            values("1, 2, 3, 4");
        }
    }
    /* receiver capacitance for condition 2 */
    receiver_capacitance() {
        when : "!I"; /* or using mode as next commented line */
        /* mode (rw, write); */
        receiver_capacitance1_rise(LTT1) {
            values("1, 2, 3, 4");
        }
        receiver_capacitance1_fall(LTT1) {
            values("1, 2, 3, 4");
        }
        receiver_capacitance2_rise(LTT1) {
            values("1, 2, 3, 4");
        }
        receiver_capacitance2_fall(LTT1) {
    }
}

```

```

        values("1, 2, 3, 4");
    }
}
pin (ZN) {
    direction : input;
    capacitance : 1.2;
...
    timing() {
...
    }
}
...
} /* end cell */
...
} /* end library */

```

Example 14-2 shows the use of the `char_when` attribute in the pin-level receiver capacitance model.

Example 14-2 Receiver Capacitance Model With the char_when Attribute

```

library (mylib) {
/* library unit, slew, OC and other library level information */
/* receiver model template */
lu_table_template ("template_8x1") {
    variable_1 : "input_net_transition";
    index_1("0, 1, 2, 3, 4, 5, 6, 7");
}
...
cell (mycell) {
    pin (Y) {
        direction : "output";
        function : "A1 * A2 * A3";
        ...
        /* timing-based ccs receiver model and driver model */
        timing () {
            related_pin : "A1";
            ...
        }
        timing () {
            related_pin : "A2";
            ...
        }
        timing () {
            related_pin : "A3";
            ...
        }
        ...
    } /* end pin Y */
    pin (A1) {
        direction : "input";
        ...
    }
}

```

```

/* pin-based ccs receiver model */
receiver_capacitance () {
    /* default condition */
    /* characterization input-state */
    char_when : "!A2 * !A3";
    receiver_capacitance1_rise ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
               0.66666, 0.77777, 0.88888");
    }
    receiver_capacitance2_rise ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
               0.66666, 0.77777, 0.88888");
    }
    receiver_capacitance1_fall ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
               0.66666, 0.77777, 0.88888");
    }
    receiver_capacitance2_fall ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
               0.66666, 0.77777, 0.88888");
    }
}
...
} /* end pin A1 */

pin (A2) {
    direction : "input";
    ...
    /* pin-based ccs receiver model */
    ...
} /* end pin A2 */

pin (A3) {
    direction : "input";
    ...

    /* pin-based CCS receiver model */
    /* enumerate all conditions */
    receiver_capacitance () {
        when : "!A1 * A2";
        receiver_capacitance1_rise ("template_8x1") {
            values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
                   0.66666, 0.77777, 0.88888");
        }
        receiver_capacitance2_rise ("template_8x1") {
            values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
                   0.66666, 0.77777, 0.88888");
        }
        receiver_capacitance1_fall ("template_8x1") {
            values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
                   0.66666, 0.77777, 0.88888");
        }
    }
}

```

```

    receiver_capacitance2_fall ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
               0.66666, 0.77777, 0.88888");
    }
}
receiver_capacitance () {
    when :"A1 * !A2";
    receiver_capacitance1_rise ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
               0.66666, 0.77777, 0.88888");
    }
    receiver_capacitance2_rise ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
               0.66666, 0.77777, 0.88888");
    }
    receiver_capacitance1_fall ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
               0.66666, 0.77777, 0.88888");
    }
    receiver_capacitance2_fall ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
               0.66666, 0.77777, 0.88888");
    }
}
receiver_capacitance () {
    when :"!A1 * !A2";
    receiver_capacitance1_rise ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
               0.66666, 0.77777, 0.88888");
    }
    receiver_capacitance2_rise ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
               0.66666, 0.77777, 0.88888");
    }
    receiver_capacitance1_fall ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
               0.66666, 0.77777, 0.88888");
    }
    receiver_capacitance2_fall ("template_8x1") {
        values("0.11111, 0.22222, 0.33333, 0.44444, 0.55555, \
               0.66666, 0.77777, 0.88888");
    }
}
...
} /* end pin A3 */
} /* end cell mycell */
} /* end library mylib */

```

Checking the receiver_capacitance Group

The Library Compiler tool automatically checks the `receiver_capacitance` group and reports any problems it encounters. For information about the types of checks the Library Compiler tool performs, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Defining the Receiver Capacitance Groups at the Timing Level

At the timing level, you do not need to define the `receiver_capacitance` group. Define the receiver capacitance for the timing arcs by using the `receiver_capacitance1_rise`, `receiver_capacitance1_fall`, `receiver_capacitance2_rise`, and `receiver_capacitance2_fall` groups in the `timing` group.

Conditional Data Modeling

Use the `mode` and `when` attributes in timing arcs for conditional timing arcs and constraints.

Defining the lu_table_template Group

Use this library-level group to create templates of common information that multiple lookup tables can use. A table template specifies the table parameters and the breakpoints for each axis. Assign each template a name so that lookup tables can refer to it.

lu_table_template Group

Define your lookup table templates in the `library` group.

lu_table_template Group Syntax

Define your lookup table templates in the `library` group.

```
library(name_id) {  
    ...  
    lu_table_template(template_name_id) {  
        variable_1: input_net_transition;  
        variable_2: total_output_net_capacitance;  
        index_1 ("float,..., float");  
        index_2 ("float,..., float");  
        ...  
    }  
    ...  
}
```

Template Variables for CCS Receiver Models

The table template specifying composite current source receiver models can have only two variables: `variable_1` and `variable_2`. The parameters are the input transition time and the total output capacitance of a constrained pin.

The index values in the `index_1` and `index_2` attributes are a list of ascending floating-point numbers.

In the timing level, the table template specifying composite current source receiver models can have two variables: `variable_1` and `variable_2`. The valid values for either variable are `input_net_transition` and `total_output_net_capacitance`.

The index values in the `index_1` and `index_2` attributes are a list of ascending floating-point numbers.

The `values` attribute defines a list of floating-point numbers that represent the capacitance for the receiver model.

`lu_table_template` Example

```
...
    lu_table_template(LTT2) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        index_1("0.1, 0.2, 0.4, 0.3");
        index_2("1.0, 2.0");
    }
```

Timing-Level receiver_capacitance Example

```
timing() /* timing arc-based receiver model*/
...
related_pin : "B"
receiver_capacitance1_rise(LTT2) {
    values("1.1, 4., 2.0, 3.2");
}
receiver_capacitance1_fall(LTT2) {
    values("1.0, 3.2, 4.0, 2.1");
}
receiver_capacitance2_rise(LTT2) {
    values("1.1, 4., 2.0, 3.2");
}
receiver_capacitance2_fall(LTT2) {
    values("1.0, 3.2, 4.0, 2.1");
}
...
}
```

Checking the Timing-Level receiver_capacitance Group

The Library Compiler tool automatically checks the `receiver_capacitance` group at the timing level and reports any problems it encounters. For information about the types of checks the Library Compiler tool performs, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Multisegment Receiver Capacitance Model

In the multisegment receiver capacitance model, the rise or fall voltage at an input or inout node is divided into N segments, where N is an integer greater than 2. For each segment, the capacitance values are stored in a lookup table using the `receiver_capacitance_rise` or `receiver_capacitance_fall` groups.

The modeling syntax is:

```
library (library_name) {
    receiver_capacitance_rise_threshold_pct ("float,...");
    receiver_capacitance_fall_threshold_pct ("float,...");
    ...
    cell(cell_name) {
        ...
        /* The receiver_capacitance group can be under a pin or timing group. */

        pin (pin_name) {
            direction : input;
            ...
            /* pin based receiver model */
            receiver_capacitance () {
                when : boolean expression;
                mode (mode_name, mode_value);
                receiver_capacitance_rise (lu_template_name) {
                    segment : "integer";
                    ...
                }
                receiver_capacitance_fall (lu_template_name) {
                    segment : "integer";
                    ...
                }
            } /* end receiver_capacitance group */
            pin (pin_name) {
                direction : output;
                ...
                timing() {
                    ...
                    when : boolean_expression;
                    mode (mode_name, mode_value);
                    receiver_capacitance_rise (lu_template_name) {
                        segment : "integer";
                        ...
                    }
                }
            }
        }
    }
}
```

```

}
receiver_capacitance_fall (lu_template_name) {
    segment : "integer";
    ...
}
} /* end timing group */
} /* end pin group */
} /* end cell group */
} /* end library group */

```

Library-Level Groups and Attributes

This section describes the library-level groups and attributes for multisegment receiver capacitance modeling.

lu_table_template Group

The `lu_table_template` group defines the template for the `receiver_capacitance_rise` and `receiver_capacitance_fall` groups specified in the `receiver_capacitance` and `timing` groups:

The template definition includes the `variable_1`, `variable_2`, `index_1`, and `index_2` attributes. The `variable_1` and `variable_2` attributes specify the index variables for the lookup tables (LUTs) in these groups. The `index_1` and `index_2` attributes specify the numerical values of the variables.

The values of `variable_1` and `variable_2` attributes are `input_net_transition` and `total_output_net_capacitance`.

You can specify the following types of lookup tables within these groups:

- One-dimensional tables with a single index, such as `input_net_transition`
- Two-dimensional tables with the indexes, `input_net_transition` and `total_output_net_capacitance`
- A scalar that has a single variation value irrespective of the `input_net_transition` and `total_output_net_capacitance` values

`receiver_capacitance_rise_threshold_pct` and `receiver_capacitance_fall_threshold_pct` Attributes

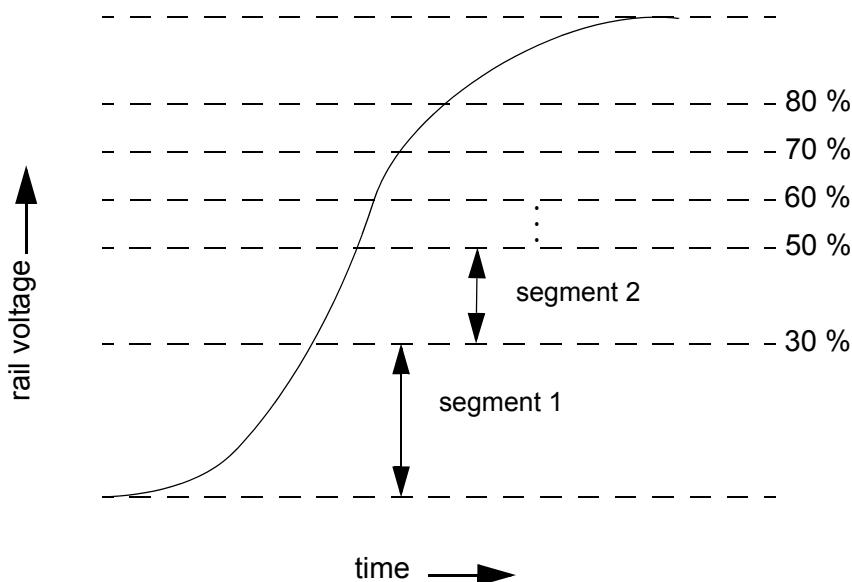
The `receiver_capacitance_rise_threshold_pct` and `receiver_capacitance_fall_threshold_pct` attributes specify the points that separate the voltage rise and fall segments. Specify each point as a percentage of the rail voltage between 0.0 and 100.0.

Specify monotonically increasing values with the `receiver_capacitance_rise_threshold_pct` attribute, and monotonically decreasing values with the `receiver_capacitance_fall_threshold_pct` attribute. For example,

```
receiver_capacitance_rise_threshold_pct ("0, 30, 50, 60, 70, 80, 100") ;
receiver_capacitance_fall_threshold_pct ("100, 80, 70, 60, 50, 30, 0") ;
```

The number of segments, N, is one less than the number of points. So, N is 6. The following figure shows the segments for a voltage rise waveform.

Figure 14-2 Segments in a Multisegment Receiver Capacitance Model



Pin and Timing Level Groups

This section describes the pin and timing level groups and attributes for multisegment receiver capacitance modeling.

receiver_capacitance Group

At the pin level, define the `receiver_capacitance_rise` and `receiver_capacitance_fall` groups in the `receiver_capacitance` group.

Each of these groups specify a lookup table. Define the template of the lookup table in the library-level `lu_table_template` group. To use the template, specify the template name as the `receiver_capacitance_rise` or `receiver_capacitance_fall` group name.

Note:

At the timing level, define the `receiver_capacitance_rise` and `receiver_capacitance_fall` groups in the timing group.

`receiver_capacitance_rise` and `receiver_capacitance_fall` Groups

The `receiver_capacitance_rise` and `receiver_capacitance_fall` groups specify the receiver capacitance values of each of the N rise and fall voltage segments, in lookup tables. Specify N `receiver_capacitance_rise` groups, each for a unique segment. The `receiver_capacitance_fall` group has the same requirement.

The `receiver_capacitance_rise` and `receiver_capacitance_fall` groups are structurally similar to the `receiver_capacitance1_rise`, `receiver_capacitance2_rise`, `receiver_capacitance1_fall`, and `receiver_capacitance2_fall` groups and include all of their attributes and subgroups.

segment Attribute

The `segment` attribute specifies the segment that the `receiver_capacitance_rise` or the `receiver_capacitance_fall` group represents. The values vary from 1 to N.

Conditional Data Modeling

You can use the `mode` and `when` attributes in both the `receiver_capacitance` and `timing` groups.

when Attribute

The `when` attribute specifies the Boolean condition for the input state of the receiver capacitance timing arc.

mode Attribute

The complex `mode` attribute specifies the current mode of the cell. When defined in the `receiver_capacitance` group, the `mode` attribute specifies the receiver capacitance for the given mode. If you specify the `mode` attribute, you must define the `mode_definition` group at the cell level.

Example

The following example shows a typical multisegment receiver capacitance model.

```
library (mylib) {
    receiver_capacitance_rise_threshold_pct ("0, 30, 50, 60, 70, 80, 100");
    receiver_capacitance_fall_threshold_pct ("100, 70, 60, 50, 40, 30, 0");
    /* receiver model template */
}
```

```

lu_table_template ("delay_template_8x1") {
    variable_1 : "input_net_transition";
    index_1("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8");
}
lu_table_template ("delay_template_8x8") {
    variable_1 : "input_net_transition";
    variable_2 : "total_output_net_capacitance";
    index_1("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8");
    index_2("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8");
}
...
cell (flop) {
    pin (D) {
        direction : input;
        /* pin-based CCS receiver model */
        receiver_capacitance () {
            /* default condition */
            /* The old model groups are shown for comparison
            receiver_capacitance1_rise(lu_template_name) {...}
            receiver_capacitance2_rise(lu_template_name) {...}
            receiver_capacitance1_fall(lu_template_name) {...}
            receiver_capacitance2_fall(lu_template_name) {...}
            */
            receiver_capacitance_rise ("delay_template_8x1") {
                segment : 1;
                values("0.11111, 0.22222, 0.33333, 0.44444, \
                        0.55555, 0.66666, 0.77777, 0.88888");
            }
            ...
            receiver_capacitance_rise ("delay_template_8x1") {
                segment : 6;
                values("0.1111, 0.2222, 0.3333, 0.4444, \
                        0.5555, 0.6666, 0.77777, 0.8888");
            }
            receiver_capacitance_fall ("delay_template_8x1") {
                segment : 1;
                values("0.11111, 0.22222, 0.33333, 0.44444, \
                        0.55555, 0.66666, 0.77777, 0.88888");
            }
            ...
            receiver_capacitance_fall ("delay_template_8x1") {
                segment : 6;
                values("0.1111, 0.2222, 0.3333, 0.4444, \
                        0.5555, 0.6666, 0.7777, 0.8888");
            }
        } /* end receiver_capacitance group */
    } /* end pin D group */
    pin (CLK) {
        direction : input;
        ...
    }
    pin (Y) {
        direction : output;
    }
}

```

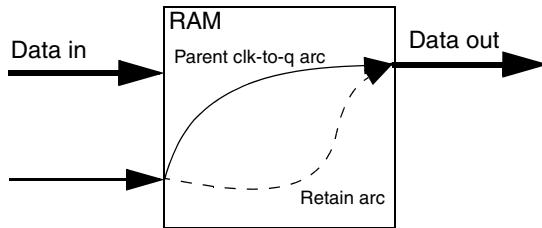
```

function : "IQ";
...
timing () {
    related_pin : "CLK";
...
/* arc-based ccs receiver model */
receiver_capacitance_rise ("delay_template_8x8") {
    segment : 1;
    values("0.11111, 0.22222, 0.33333, 0.44444, \
           0.55555, 0.66666, 0.77777, 0.88888, ...");
}
...
receiver_capacitance_rise ("delay_template_8x8") {
    segment : 6;
    values("0.11111, 0.22222, 0.33333, 0.44444, \
           0.55555, 0.66666, 0.77777, 0.88888, ...");
}
receiver_capacitance_fall ("delay_template_8x8") {
    segment : 1;
    values("0.11111, 0.22222, 0.33333, 0.44444, \
           0.55555, 0.66666, 0.77777, 0.88888, ...");
}
...
receiver_capacitance_fall ("delay_template_8x8") {
    segment : 6;
    values("0.11111, 0.22222, 0.33333, 0.44444, \
           0.55555, 0.66666, 0.77777, 0.88888, ...");
}
} /* end timing group */
} /* end pin Y group*/
} /* end cell flop group*/
} /* end library mylib */

```

CCS Retain Arc Support

A *retain delay* is the shortest delay among all parallel arcs, from the input port to the output port. *Access time* is the longest delay from the input port to the output port. The output value is uncertain and unstable in the time interval between the retain delay and the access time. A retain arc, as shown in [Figure 14-3](#), ensures that the output does not change during this time interval.

Figure 14-3 Retain Arc Example

Retain arcs:

- Guarantee that the output does not change for a certain time interval.
- Are usually defined for memory cells.
- Are not inferred as a timing check but are inferred as a delay arc.

Liberty syntax supports retain arcs in nonlinear delay models by providing the following timing groups:

- retaining_rise
- retaining_fall
- retain_rise_slew
- retain_fall_slew

CCS Retain Arc Syntax

The following syntax supports CCS timing models including the expanded and compact CCS timing models. Because retain arcs have no relation to CCS receiver models, only syntax for CCS driver models is described as follows:

Syntax

```
library (library_name) {
    delay_model : table_lookup;
    ...
    output_current_template(template_name) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        variable_3: time;
        index_1("float, ..., float"); /* optional at library level */
        index_2("float, ..., float"); /* optional at library level*/
        index_3("float, ..., float"); /* optional at library level*/
    }
    cell(name) {
```

```

pin (name) {
    timing() {
        ccs_retain_rise() {
            vector(template_name) {
                reference_time : float;
                index_1("float");
                index_2("float");
                index_3("float, ..., float");
                values("float, ..., float");
            }
            vector(template_name) { . . . } ...
        }
        ccs_retain_fall() {
            vector(template_name) {
                reference_time : float;
                index_1("float");
                index_2("float");
                index_3("float, ..., float");
                values("float, ..., float");
            }
            vector(template_name) { . . . } ...
        }
        output_current_rise() { ... }
        output_current_fall() { ... }
    }
}
. . .
}

```

The format of the expanded CCS retain arc group is the same as the general CCS timing arcs that are defined by using the `output_current_rise` and `output_current_fall` groups.

ccs_retain_rise and ccs_retain_fall Groups

The `ccs_retain_rise` and `ccs_retain_fall` groups are provided in the timing group for expanded CCS retain arcs.

vector Group

The current `vector` group in the `ccs_retain_rise` and `ccs_retain_fall` groups uses the lookup table template defined by `output_current_template`. The `vector` group has the following parameters:

- `input_net_transition`
- `total_output_net_capacitance`
- `time`

For every value pair (such as `input_net_transition` and `total_output_net_capacitance`), there is a specified `current(time)` vector.

reference_time Attribute

The `reference_time` simple attribute specifies the time that the input signal waveform crosses the rising or falling input delay threshold.

Compact CCS Retain Arc Syntax

The compact CCS retain arc format is the same as a general compact CCS timing arc. The following retain arc syntax supports compact CCS timing.

Syntax

```
library(my_lib) {
...
base_curves (base_curves_name) {
    base_curve_type: enum (ccs_timing_half_curve);
    curve_x ("float..., float");
    curve_y (integer, "float..., float");
    curve_y (integer, "float..., float");
    ...
}
compact_lut_template(template_name) {
    base_curves_group: base_curves_name;
    variable_1 : input_net_transition;
    variable_2 : total_output_net_capacitance;
    variable_3 : curve_parameters;
    index_1 ("float..., float");
    index_2 ("float..., float");
    index_3 ("string..., string");
}
...
cell(cell_name) {
...
pin(pin_name) {
    direction : string;
    capacitance : float;
    timing() {
        compact_ccs_retain_rise (template_name) {
            base_curves_group : "base_curves_name";
            index_1 ("float..., float");
            index_2 ("float..., float");
            index_3 ("string..., string");
            values ("..."...)
        }
        compact_ccs_retain_fall (template_name) {
...
}
}
}
```

```

base_curves_group : "base_curves_name";
index_1 ("float..., float");
index_2 ("float..., float");
index_3 ("string..., string");
values ("..."...)
}
compact_ccs_rise(template_name) { ... }
compact_ccs_fall(template_name) { ... }
. . .
}/*end of timing */
}/*end of pin */
}/*end of cell */
...
}/* end of library*/

```

compact_ccs_retain_rise and compact_ccs_retain_fall Groups

The `compact_ccs_retain_rise` and `compact_ccs_retain_fall` groups are provided in the timing group for compact CCS retain arcs.

base_curves_group Attribute

The `base_curves_group` attribute is optional. The attribute is required when `base_curves_name` is different from that defined in the `compact_lut_template` template name.

index_1, index_2, and index_3 Attributes

The values for the `index_1` and `index_2` attributes are `input_net_transition` and `total_output_net_capacitance`.

The values for the `index_3` attribute must contain the following base curve parameters:

- `init_current`
- `peak_current`
- `peak_voltage`
- `peak_time`
- `left_id`
- `right_id`

values Attribute

The `values` attribute provides the compact CCS retain arc data values. The `left_id` and `right_id` values for compact CCS timing base curves should be integers, and they must be predefined in the `base_curves` group.

Composite Current Source Driver and Receiver Model Example

[Example 14-3](#) is an example of composite current source driver and receiver model syntax.

Example 14-3 Composite Current Source Driver and Receiver Model

```
library(new_lib) {
    .
    .
    output_current_template(CCT) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        variable_3: time;
    }
    lu_table_template(LTT1) {
        variable_1: input_net_transition;
        index_1("0.1, 0.2, 0.3, 0.4");
    }
    lu_table_template(LTT2) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        index_1("1.1, 2.2");
        index_2("1.0, 2.0");
    }
    .
    .
    cell(DFD) {
        pin(D) /* pin-based receiver model*/
            direction : input;
            receiver_capacitance() {
                receiver_capacitance1_rise(LTT1) {
                    values("1.1, 0.2, 1.3, 0.4");
                }
                receiver_capacitance1_fall(LTT1) {
                    values("1.0, 2.1, 1.3, 1.2");
                }
                receiver_capacitance2_rise(LTT1) {
                    values("0.1, 1.2, 0.4, 1.3");
                }
                receiver_capacitance2_fall(LTT1) {
                    values("1.4, 2.3, 1.2, 1.1");
                }
            }
        }/*end of pin (D)*/
    } /*end of cell (DFD)*/
    .
    .
    cell() {
        .
        .
    }
}
```

```
pin (Y) {
    direction : output;
    capacitance : 1.2;

    timing() /* CCS and arc-based receiver model */

    . .
    related_pin : "B";
    receiver_capacitance1_rise(LTT2) {
        values("0.1, 1.2");
        values("3.0, 2.3");
    }
    receiver_capacitance1_fall(LTT2) {
        values("1.1, 2.3");
        values("1.3, 0.4");
    }
    receiver_capacitance2_rise(LTT2) {
        values("1.3, 0.2");
        values("1.3, 0.4");
    }
    receiver_capacitance2_fall(LTT2) {
        values("1.3, 2.1");
        values("0.4, 1.3");
    }
    output_current_rise() {
        vector(CCT) {
            reference_time : 0.05;
            index_1(0.1);
            index_2(1.0);
            index_3("1.0, 1.5, 2.0, 2.5, 3.0");
            values("1.1, 1.2, 1.5, 1.3, 0.5");
        }
        vector(CCT) {
            reference_time : 0.05;
            index_1(0.1);
            index_2(2.0);
            index_3("1.2, 2.2, 3.2, 4.2, 5.2");
            values("1.11, 1.31, 1.51, 1.41, 0.51");
        }
        vector(CCT) {
            reference_time : 0.06;
            index_1(0.2);
            index_2(1.0);
            index_3("1.2, 2.1, 3.2, 4.2, 5.2");
            values("1.0, 1.5, 2.0, 1.2, 0.4");
        }
        vector(CCT) {
            reference_time : 0.06;
            index_1(0.2);
            index_2(2.0);
            index_3("1.2, 2.2, 3.2, 4.2, 5.2");
            values("1.11, 1.21, 1.51, 1.41, 0.31");
        }
    }
}
```

```

}
output_current_fall() {
    vector(CCT) {
        reference_time : 0.05;
        index_1(0.1);
        index_2(1.0);
        index_3("0.1, 2.3, 3.3, 4.4, 5.0");
        values("-1.1, -1.3, -1.6, -1.4, -0.5");
    }
    vector(CCT) {
        reference_time : 0.05;
        index_1(0.1);
        index_2(2.0);
        index_3("1.2, 2.2, 3.2, 4.2, 5.2");
        values("1.11, -1.21, -1.41, -1.31, -0.51");
    }
    vector(CCT) {
        reference_time : 0.13;
        index_1(0.2);
        index_2(1.0);
        index_3("0.1, 1.3, 2.3, 3.4, 5.0");
        values("-1.1, -1.3, -1.8, -1.4, -0.5");
    }
    vector(CCT) {
        reference_time : 0.13;
        index_1(0.2);
        index_2(2.0);
        index_3("1.2, 2.2, 3.2, 4.2, 5.2");
        values("-1.11, -1.31, -1.81, -1.51, -0.41");
    }
}/*end of timing*/
.
.
.
} /* end of pin (Y) */
.
.
.
} /* end of cell */
.
.
.
} /* end of library */

```

Checking CCS Library Models

The Library Compiler tool automatically checks CCS models, including the polarity of the current and the voltage swing, and reports any problems it encounters. For information about the types of checks the Library Compiler tool performs for CCS models, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

15

Advanced Composite Current Source Modeling

This chapter provides an overview of advanced composite current source (CCS) modeling to support nanometer and very deep submicron IC development. The following composite current source modeling topics are covered:

- [Modeling Cells With Advanced Composite Current Source Information](#)
- [Compact CCS Timing Model](#)

Modeling Cells With Advanced Composite Current Source Information

Composite current source modeling supports additional driver model complexity by using a time- and voltage- dependent current source with essentially an infinite drive resistance. To achieve high accuracy, this driver model maps the transistor behavior for lumped loads to that for parasitic network instead of modeling the transistor behavior.

The composite current source model has better receiver model accuracy because the input capacitance of a receiver is dynamically adjusted during the transition by using two capacitance values. You can use the driver model with or without the receiver model.

Compact CCS Timing Model

Conventional CCS timing driver models require you to describe each CCS driver switching current waveform by sampling data points. As the number of timing arcs increase in a standard cell library, the CCS timing library size can become very large.

This section describes the syntax of a compact model that uses indirectly shared base curves to model the shape of switching curves. By allowing each base curve to model multiple switching curves with similar shapes, the modeling efficiency is improved and the library size is compressed.

The topics in the following sections include:

- Describing CCS timing base curves.
- Describing the syntax of base curves and the compact CCS driver modeling format.
- Describing Library Compiler screener rules for reading compact CCS timing data into the Library Compiler tool.

Modeling With CCS Timing Base Curves

CCS driver switching curves in the I-V domain are smoother than those in the I(t) and V(t) domains. The I-V switching curves are usually convex, and they have no inflection point in the middle, a feature that facilitates compact modeling.

[Figure 15-1](#) illustrates modeling an inverter cell rise transition with the existing CCS format. The figure shows the I(t) curve, corresponding V(t) curve, and I-V curves.

The CCS segmentation process adaptively samples nine data points from the I(t) curve based on the given tolerance. There are 18 floating-point numbers (nine time points + nine current points) that are stored in the CCS library.

Figure 15-1 Inverter Cell Rise Transition With Existing CCS Format

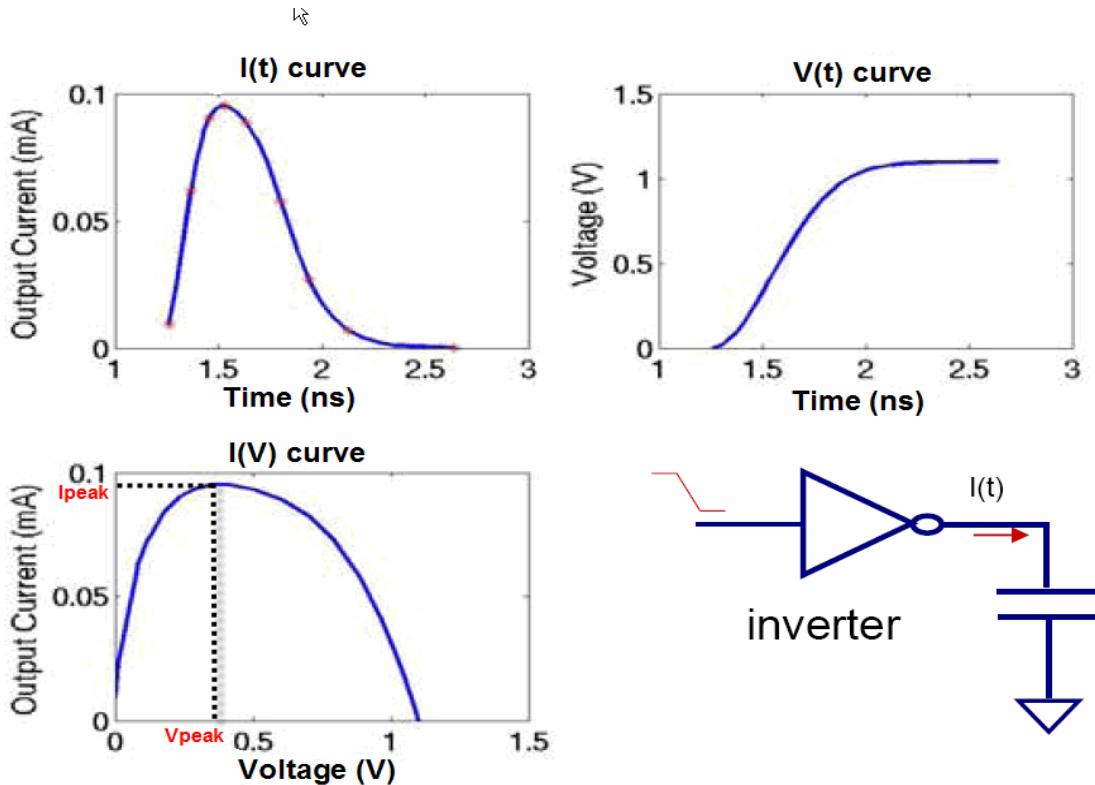


Figure 15-2 shows the mapping of I-V curves to base curves. The I-V curve is split into two halves at the peak, and an eleven point normalized base curve in the base curve database is selected to exactly match each half curve.

Only six parameters are required to model this inverter switching curve, reducing the storage cost by three times. The six parameters are as follows:

linit

Switching current value at the starting point.

Ipeak

Peak switching current value.

Vpeak

Voltage value when current reaches peak value.

Tpeak

Time when current reaches peak value.

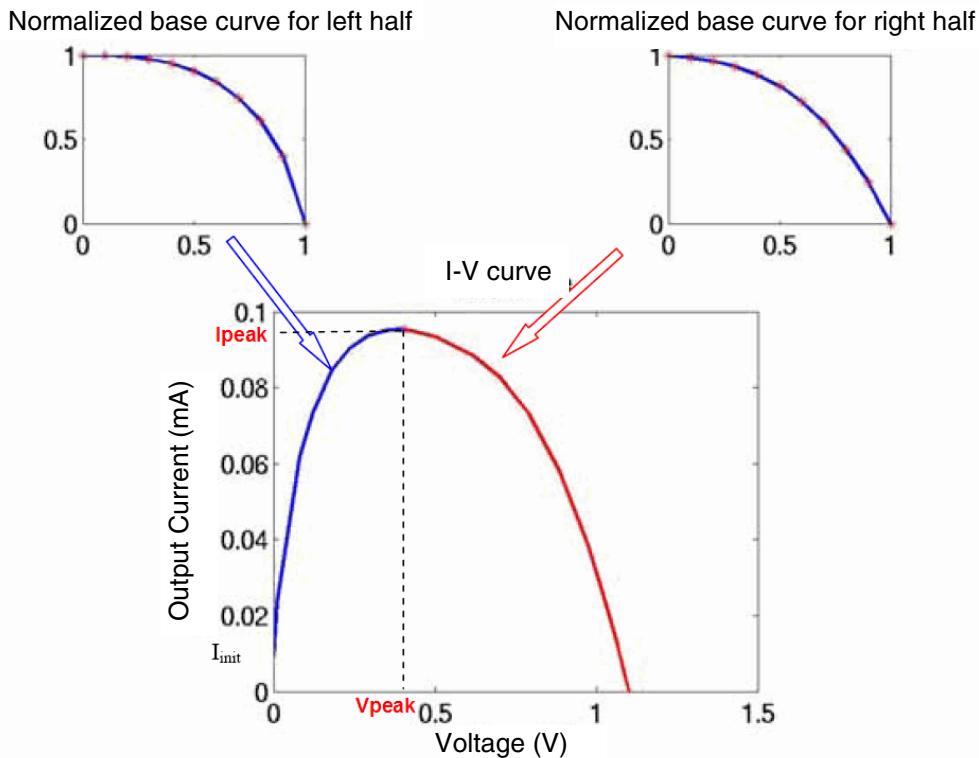
Left id

Reference id of the base curve that matches the left half.

Right id

Reference id of the base curve that matches the right half.

Figure 15-2 Using Base Curves to Simplify I-V Curve Modeling



Compact CCS Timing Model Syntax

In Figure 15-2, each switching I-V curve can be modeled by normalized base curves using the following parameters: `init_current`, `peak_current`, `peak_voltage`, `peak_time`, `left_id`, and `right_id`.

The `init_current`, `peak_current`, `peak_voltage`, and `peak_time` parameters are the critical characteristics of the curve. The `left_id` and `right_id` describe the two base curves that represent the two halves of the I-V curve.

The syntax for compact CCS timing model is as follows:

```
library(my_compact_ccs_lib) {
...
}
```

```

base_curves (base_curves_name) {
    base_curve_type: enum (ccs_timing_half_curve);
    curve_x ("float..., float");
    curve_y (curve_id, "float..., float");
    curve_y (curve_id, "float..., float");
    ...
    curve_y (curve_id, "float..., float");
}

compact_lut_template(template_name) {
    base_curves_group: base_curves_name;
    variable_1 : input_net_transition |
    total_output_net_capacitance;
    variable_2 : input_net_transition |
    total_output_net_capacitance;
    variable_3 : curve_parameters;
    index_1 ("float..., float");
    index_2 ("float..., float");
    index_3 ("string..., string");
}
...
cell(cell_name) {
...
pin(pin_name) {
    direction : string;
    capacitance : float;
    timing() {
        /*Compact CCS arc-based driver model*/
        compact_ccs_rise (template_name) {
            base_curves_group : "base_curves_name";
            values (...", "... \
            "...", "... \
            "...", "... \
            "...", ...)
        }/*end of compact_ccs_rise() */
        compact_ccs_fall(template_name) {
            ...
        }/*end of compact_ccs_fall() */
        ...
    } /*end of timing */
} /*end of pin */
} /*end of cell */
...
} /* end of library*/

```

The groups described in the following sections support compact CCS timing models.

base_curves Group

The `base_curves` library-level group contains the detailed description of normalized base curves. The `base_curves` group has the following attributes:

`base_curve_type` Complex Attribute

The `base_curve_type` attribute specifies the type of base curve. The only valid value for this attribute is `ccs_timing_half_curve`.

`curve_x` Complex Attribute

The data array contains the x-axis values of the normalized base curve. Only one `curve_x` value is allowed for each `base_curves` group. See [Figure 15-2](#) for more details.

For a `ccs_timing_half_curve` type base curve, the `curve_x` value must be between 0 and 1 and increase monotonically.

`curve_y` Complex Attribute

Each base curve (as illustrated in [Figure 15-2](#)) is composed of one `curve_x` and one `curve_y`. You should define the `curve_x` base curve before `curve_y` for better clarity and easier implementation.

There are two data sections in the `curve_y` complex attribute:

- `curve_id` is the identifier of the base curve.
- Data array is the y-axis values of the normalized base curve.

compact_lut_template Group

The `compact_lut_template` group is used for compact CCS timing modeling. (The `lu_table_template` group is used for other timing models.) The `compact_lut_template` group has the following attributes:

`base_curves_group` Attribute

This is a required attribute in the `compact_lut_template` group. Its value should be a predefined `base_curves` group name. The `base_curve_type` of `base_curves` group determines the values in `index_3`. It also determines where you can use this template.

`variable_1` and `variable_2` Attributes

Only `input_net_transition` and `total_output_net_capacitance` are valid values for `variable_1` and `variable_2`.

variable_3 Attribute

Only `curve_parameters` is a string value for this variable.

index_1 and index_2 Attributes

These are required attributes. They define the `input_net_transition` or `total_output_net_capacitance` values using the float notation.

index_3 Attribute

String values in `index_3` are determined by the `base_curve_type` in `base_curves` group. When the `base_curve_type` is a `ccs_timing_half_curve`, at least six string parameters should be defined. They are `init_current`, `peak_current`, `peak_voltage`, `peak_time`, `left_id`, and `right_id`. If any of these six parameters are missing, a compilation error is issued.

The Library Compiler tool allows more than six parameters for cases that require more parameters to describe the original data (for example, Vddcell and Vsscell, or the final voltage of load capacitor in cell rise or fall transition).

compact_ccs_{rise|fall} Group

This is the compact CCS timing data in timing arc group, which has the following attributes:

base_curves_group Attribute

Defining this attribute is optional when the `base_curves_name` is same as that defined in the `compact_lut_template` group. This group is referenced by the `compact_ccs_{rise|fall}` group.

values Attribute

Values of compact CCS timing data depend on how you define `index_3` values. The Library Compiler tool checks the data specified in values according to the string order of `index_3`.

For compact CCS timing, base curves data `left_id` and `right_id` values can only be integers. If more than, six parameters are specified in `index_3`, the Library Compiler tool does not check for other data in the values group. These six parameters are stored in the database as is.

Checking the compact_lut_template Group

The Library Compiler tool automatically checks the `compact_lut_template` group and issues a compilation error message if it encounters problems. For information about the types of checks the Library Compiler tool performs for the `compact_lut_template` group, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

CCS Timing Library Example

The following example shows the CCS timing library with the compact syntax:

```
library(my_lib) {
...
/* normal lu table template for timing arcs */
lu_table_template (LTT2) {
    variable_1 : input_net_transition;
    variable_2 : total_output_net_capacitance;
    index_1 ("0.1, 0.2");
    index_2 ("1.0, 2.0");
}
base_curves (ctbctl1){
    base_curve_type : ccs_timing_half_curve;
    curve_x("0.2, 0.5, 0.8");
    curve_y(1, "0.8, 0.5, 0.2");
    curve_y(2, "0.75, 0.5, 0.35");
    ...
    curve_y(100, "0.85, 0.5, 0.15");
}
...
/* New lu table template for compact CCS timing model*/
compact_lut_template(LTT3) {
    variable_1 : input_net_transition;
    variable_2 : total_output_net_capacitance;
    variable_3 : curve_parameters;
    index_1 ("0.1, 0.2");
    index_2 ("1.0, 2.0");
    index_3 ("init_current, peak_current, peak_voltage, peak_time, left_id,
right_id");
    base_curves_group: "ctbctl1";
}
...
cell(cell_name) {
...
pin(Y) {
direction : output;
capacitance : 1.2;
timing() {

```

```
/*compact CCS and arc-based receiver model*/
compact_ccs_rise(LTT3) {
    base_curves_group : "ctbct1"; /* optional*/
    values("0.1, 0.5, 0.6, 0.8, 1, 3", \
    "0.15, 0.55, 0.65, 0.85, 2, 4", \
    "0.2, 0.6, 0.7, 0.9, 3, 2", \
    "0.25, 0.65, 0.75, 0.95, 4, 1")
}/*end of compact_ccs_rise() */
compact_ccs_fall(LTT3) {
    .
    .
}/*end of compact_ccs_fall() */
.
.
}/*end of timing */
}/*end of pin(Y) */
}/*end of cell */
. . .
}/* end of library*/
```


16

Composite Current Source Signal Integrity Modeling

This chapter provides an overview of composite current source (CCS) modeling to support noise (signal integrity) modeling for advanced technologies. This chapter includes the following sections:

- [CCS Signal Integrity Modeling Overview](#)
- [CCS Noise Modeling for Unbuffered Cells With a Pass Gate](#)
- [CCS Noise Modeling for Multivoltage Designs](#)
- [Referenced CCS Noise Modeling](#)

CCS Signal Integrity Modeling Overview

CCS noise modeling can capture essential noise properties of digital circuits using a compact library representation. It enables fast and accurate gate-level noise analysis while maintaining a relatively simple library characterization. CCS noise modeling supports noise combination and driver weakening.

CCS noise is characterization data that provides information for noise failure detection on cell inputs, calculation of noise bumps on cell outputs, and noise propagation through the cell. For the best accuracy, you must add CCS timing data to the library in addition to the CCS noise data. The CCS noise data includes the following:

- Channel-connected block parameters
- DC current tables
- Timing tables for rising and falling transitions
- Timing tables for low and high propagated noise

Compiling a Library With CCS Signal Integrity Information

When you compile a library containing CCS noise information with the `read_lib` command in the Library Compiler tool (after screening data without finding any errors), the Library Compiler tool extracts the required information from the CCS noise data in the library.

Often you only need to screen the CCS noise information in the libraries to see if the data is acceptable. This involves including or excluding the noise compilation. The `lc_disable_ccs_noise_extraction` environment variable enables you to turn on or off the compilation of CCS noise information.

By default, the `lc_disable_ccs_noise_extraction` variable is set to 0, which means that the Library Compiler tool screens and extracts CCS noise information in the libraries with the `read_lib` command.

If the `lc_disable_ccs_noise_extraction` variable is set to 1 before reading in a library with the `read_lib` command, the CCS noise information is screened but not extracted into the compiled library database. As a result, the compiled library database does not contain the CCS noise information.

CCS Signal Integrity Modeling Syntax

```
library (name) {  
  ...  
  lu_table_template(dc_template_name) {  
    variable_1 : input_voltage;
```

```

        variable_2 : output_voltage;
    }
lu_table_template(output_voltage_template_name) {
    variable_1 : input_net_transition;
    variable_2 : total_output_net_capacitance;
    variable_3 : time;
}
lu_table_template(propagated_noise_template_name) {
    variable_1 : input_noise_height;
    variable_2 : input_noise_width;
    variable_3 : total_output_net_capacitance;
    variable_4 : time;
}
cell (name) {
    pin (name) {
        ...
        ccsn_first_stage () {
            is_needed : true | false;
            is_inverting : Boolean;
            stage_type : stage_type_value;
            miller_cap_rise : float;
            miller_cap_fall : float;
            output_signal_level : power_supply_name;
            dc_current (dc_current_template) {
                index_1("float, ...");
                index_2("float, ...");
                values("float, ...");
            }
            output_voltage_rise ( )
                vector (output_voltage_template_name) {
                    index_1(float);
                    index_2(float);
                    index_3("float, ...");
                    values("float, ...");
                }
            ...
        }
        output_voltage_fall ( ) {
            vector (output_voltage_template_name) {
                index_1(float);
                index_2(float);
                index_3("float, ...");
                values("float, ...");
            }
        ...
    }
    propagated_noise_low ( ) {
        vector (propagated_noise_template_name) {
            index_1(float);
            index_2(float);
            index_3(float);
            index_4("float, ...");
    }
}

```

```
        values("float, ...");
    }
    ...
}
propagated_noise_high () {
    vector (propagated_noise_template_name) {
        index_1(float);
        index_2(float);
        index_3(float);
        index_4("float, ...");
        values("float, ...");
    }
    ...
}
when : "Boolean expression";
} /* ccsn_first_stage */
ccsn_last_stage () {
    is_needed : true | false;
    is_inverting : Boolean;
    stage_type : stage_type_value;
    miller_cap_rise : float;
    miller_cap_fall : float;
    input_signal_level : power_supply_name;
    dc_current (dc_current_template) {
        index_1("float, ...");
        index_2("float, ...");
        values("float, ...");
    }
    output_voltage_rise () {
        vector (output_voltage_template_name) {
            index_1(float);
            index_2(float);
            index_3("float, ...");
            values("float, ...");
        }
        ...
    }
    output_voltage_fall () {
        vector (output_voltage_template_name) {
            index_1(float);
            index_2(float);
            index_3("float, ...");
            values("float, ...");
        }
        ...
    }
}
propagated_noise_low () {
    vector (propagated_noise_template_name) {
        index_1(float);
        index_2(float);
        index_3(float);
        index_4("float, ...");
        values("float, ...");
    }
}
```

```

        }
        ...
    }
    propagated_noise_high () {
        vector (propagated_noise_template_name) {
            index_1(float);
            index_2(float);
            index_3(float);
            index_4("float, ...");
            values("float, ...");
        }
        ...
    }
    when : "Boolean expression";
} /* ccsn_last_stage */
...
timing() {
    ...
    ccsn_first_stage () {
        is_needed : true | false;
        is_inverting : Boolean;
        stage_type : stage_type_value;
        miller_cap_rise : float;
        miller_cap_fall : float;
        output_signal_level : power_supply_name;
        dc_current (dc_current_template) {
            index_1("float, ...");
            index_2("float, ...");
            values("float, ...");
        }
        output_voltage_rise () {
            vector (output_voltage_template_name) {
                index_1(float);
                index_2(float);
                index_3("float, ...");
                values("float, ...");
            }
            ...
        }
        output_voltage_fall () {
            vector (output_voltage_template_name) {
                index_1(float);
                index_2(float);
                index_3("float, ...");
                values("float, ...");
            }
            ...
        }
    }
    propagated_noise_low () {
        vector (propagated_noise_template_name) {
            index_1(float);

```

```
    index_2(float);
    index_3(float);
    index_4("float, ...");
    values("float, ...");
}
...
}
propagated_noise_high () {
    vector (propagated_noise_template_name) {
        index_1(float);
        index_2(float);
        index_3(float);
        index_4("float, ...");
        values("float, ...");
    }
...
}
when : "Boolean expression";
} /* ccsn_first_stage */
ccsn_last_stage () {
    is_needed : true | false;
    is_inverting : Boolean;
    stage_type : stage_type_value;
    miller_cap_rise : float;
    miller_cap_fall : float;
    input_signal_level : power_supply_name;
    dc_current (dc_current_template)
        index_1("float, ...");
        index_2("float, ...");
        values("float, ...");
}
output_voltage_rise ()
    vector (output_voltage_template_name) {
        index_1(float);
        index_2(float);
        index_3("float, ...");
        values("float, ...");
    }
...
}
output_voltage_fall ()
    vector (output_voltage_template_name) {
        index_1(float);
        index_2(float);
        index_3("float, ...");
        values("float, ...");
    }
...
}
propagated_noise_low ()
    vector (propagated_noise_template_name) {
```

```

        index_1(float);
        index_2(float);
        index_3(float);
        index_4("float, ...");
        values("float, ...");
    }
    ...
}
propagated_noise_high () {
    vector (propagated_noise_template_name) {
        index_1(float);
        index_2(float);
        index_3(float);
        index_4("float, ...");
        values("float, ...");
    }
    ...
}
when : "Boolean expression";
} /* ccsn_last_stage */
} /* end timing/pin/cell/library group */

```

Library-Level Groups and Attributes

This section describes the library-level groups and attributes used for CCS noise modeling.

lu_table_template Group

The `lu_table_template` group creates the lookup-table template for the `dc_current` group and vectors for the `output_voltage_rise`, `output_voltage_fall`, `propagated_noise_high`, and `propagated_noise_low` groups.

variable_1, variable_2, variable_3, and variable_4 Attributes

Set the `variable_1`, `variable_2`, `variable_3`, and `variable_4` attributes inside the `lu_table_template` group.

You can specify the template used for the following tables and vectors by using a combination of these attributes:

- The `output_current_rise` and `output_current_fall` group vectors
Valid values for `variable_1`, `variable_2`, and `variable_3` are `input_net_transition`, `total_output_net_capacitance`, and `time`, respectively.
- The `propagated_noise_low` and `propagated_noise_high` group vectors

Valid values for variable_1, variable_2, variable_3, and variable_4 are input_noise_height, input_noise_width, total_output_net_capacitance, and time, respectively.

- The template used for the `dc_current` tables

Valid values for variable_1 and variable_2 are input_voltage and output_voltage, respectively.

Pin-Level Groups and Attributes

This section describes the pin-level groups and attributes used for CCS noise modeling.

`ccsn_first_stage` and `ccsn_last_stage` Groups

The `ccsn_first_stage` and `ccsn_last_stage` groups specify CCS noise data for the first stage or the last stage of channel-connected blocks. The `ccsn_first_stage` and `ccsn_last_stage` groups can be defined inside timing or pin groups.

The `ccsn_first_stage` and `ccsn_last_stage` groups contain the following:

- The `is_needed`, `is_inverting`, `stage_type`, `miller_cap_rise` and `miller_cap_fall`, and `output_signal_level` or `input_signal_level` channel-connected block attributes
- The `dc_current` group, which contains a two-dimensional DC current table
- The `output_current_rise` and `output_current_fall` groups, which contain two timing tables for rising and falling transitions.
- The `propagated_noise_low` and `propagated_noise_high` groups, which contain two noise tables for low and high propagated noise.

Note:

If the `ccsn_first_stage` and `ccsn_last_stage` groups are defined at the pin level, the `ccsn_first_stage` group can be defined only in an input pin or inout pin, and the `ccsn_last_stage` group can be defined only in an output pin or inout pin.

`is_needed` Attribute

The `is_needed` Boolean attribute determines whether the `dc_current`, `output_current_rise`, `output_current_fall`, `propagated_noise_low`, and `propagated_noise_high` channel-connected block attributes should be specified to include CCS noise data for a cell. The `is_needed` attribute is defined inside the `ccsn_first_stage` and `ccsn_last_stage` groups.

By default, the `is_needed` attribute is set to `true`, which means that CCS noise data is included in the `ccsn_first_stage` and `ccsn_last_stage` groups for the cell. The `is_needed` attribute should be set to `false` for cells that do not need a current-based driver model, such as diodes, antennas, and cload cells. When the attribute is set to `false`, CCS noise data, enabled by the channel-connected block attributes, is not included in the `ccsn_first_stage` and `ccsn_last_stage` groups.

is_inverting Attribute

The `is_inverting` attribute specifies whether the channel-connected block is inverting. If the channel-connected block is inverting, set the `is_inverting` attribute to `true`. Otherwise, set the attribute to `false`. This attribute is mandatory if the `is_needed` attribute is set to `true`. Note that the `is_inverting` attribute is different from the “invertness” or `timing_sense` of the timing arc, which might consist of multiple channel-connected blocks.

stage_type Attribute

The `stage_type` attribute specifies the channel-connected block’s output voltage stage type. The valid values are `pull_up`, which causes the channel-connected block’s output voltage to be pulled up or to rise; `pull_down`, which causes the channel-connected block’s output voltage to be pulled down or to fall; and `both`, which causes the channel-connected block’s output voltage to be pulled up or down.

miller_cap_rise and miller_cap_fall Attributes

The `miller_cap_rise` and `miller_cap_fall` float attributes specify the Miller capacitance value for a rising and falling channel-connected block output transition. The value must be greater than or equal to zero. The attributes are defined inside the `ccsn_first_stage` and `ccsn_last_stage` groups.

output_signal_level and input_signal_level Attributes

The `output_signal_level` and `input_signal_level` attributes specify the power supply voltage names for a channel-connected block output and input, respectively. The `output_signal_level` attribute is defined inside the `ccsn_first_stage` group and the `input_signal_level` attribute is defined inside the `ccsn_last_stage` group.

Note:

The `output_signal_level` and `input_signal_level` attribute specifications within the `ccsn_first_stage` and `ccsn_last_stage` groups override the `output_signal_level` and `input_signal_level` attribute specifications at the pin level.

For a timing arc, the `output_signal_level` attribute specification within the `ccsn_first_stage` group overrides the `output_signal_level` attribute specification for the related pin (defined by the `related_pin` attribute).

dc_current Group

The `dc_current` group specifies the input and output voltage values of a two-dimensional current table for a channel-connected block. Use the `index_1` and `index_2` attributes, respectively, to list the input and output voltage values in library voltage units. Specify the `values` attribute in the `dc_current` group to list the relative channel-connected block DC current values, in library current units, that are measured at the channel-connected block output node.

output_voltage_rise and output_voltage_fall Groups

The `output_voltage_rise` and `output_voltage_fall` groups specify `vector` groups that describe three-dimensional `output_voltage` tables for a channel-connected block whose output node's voltage values are rising or falling. The groups are defined inside the `ccsn_first_stage` and `ccsn_last_stage` groups.

Specify the following attributes in the `vector` group: The `index_1` attribute lists the `input_net_transition` (slew) values in library time units. The `index_2` attribute lists the `total_output_net_capacitance` (load) values in library capacitance units. The `index_3` attribute lists the sampling time values in library time units. The `values` attribute lists the voltage values, in library voltage units, that are measured at the channel-connected block output node.

propagated_noise_low and propagated_noise_high Groups

The `propagated_noise_low` and `propagated_noise_high` groups use `vector` groups to specify the three-dimensional `output_voltage` tables of the channel-connected block whose output node's voltage values are rising or falling. The groups are defined inside the `ccsn_first_stage` and `ccsn_last_stage` groups.

Specify the following attributes in the `vector` group: The `index_1` attribute lists the `input_noise_height` values in library voltage units. The `index_2` attribute lists the `input_noise_width` values in library time units. The `index_3` attribute lists the `total_output_net_capacitance` values in library capacitance units. The `index_4` attribute lists the sampling time values in library time units. The `values` attribute lists the voltage values, in library voltage units, that are measured at the channel-connected block output node.

when Attribute

The `when` attribute specifies the condition under which the channel-connected block data is applied. The attribute is defined in the `ccsn_first_stage` and `ccsn_last_stage` groups both at the pin level and the timing level.

Checking CCS Signal Integrity Models

The Library Compiler tool automatically checks the syntax for CCS signal integrity models and reports any problems it encounters. For information about the types of checks the Library Compiler tool performs for CCS signal integrity models, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

CCS Noise Library Example

The following is an example CCS noise library.

Example 16-1 CCS Noise Library

```
library (CCS_noise) {

    technology ( cmos ) ;
    delay_model      : table_lookup;
    time_unit        : "1ps" ;
    leakage_power_unit : "1pW" ;
    voltage_unit     : "1V" ;
    current_unit     : "1uA" ;
    pulling_resistance_unit : "1kohm" ;
    capacitive_load_unit(1000.000,ff) ;

    nom_voltage      : 1.200;
    nom_temperature  : 25.000;
    nom_process       : 1.000;

    operating_conditions("OC1") {
        process : 1.000;
        temperature : 25.000;
        voltage : 1.200;
        tree_type : "balanced_tree";
    }
    default_operating_conditions:OC1;

    lu_table_template(del_0_5_7_t) {
        variable_1 : input_net_transition;
        index_1("10.000, 175.000, 455.000, 980.000, 2100.000");
        variable_2 : total_output_net_capacitance;
        index_2("0.000000, 0.004000, 0.007000, 0.019000, 0.040000, 0.075000,\n
                0.175000");
    }
}
```

```

lu_table_template(ccsn_dc_29x29) {
    variable_1 : input_voltage;
    variable_2 : output_voltage;
}

lu_table_template(ccsn_timing_lut_5) {
    variable_1 : input_net_transition;
    variable_2 : total_output_net_capacitance;
    variable_3 : time;
}

lu_table_template(ccsn_prop_lut_5) {
    variable_1 : input_noise_height;
    variable_2 : input_noise_width;
    variable_3 : total_output_net_capacitance;
    variable_4 : time;
}

lu_table_template(lu_table_template7x9) {
    variable_1 : input_net_transition;
    variable_2 : voltage;
}
cell(inv) {
    area : 0.75;
    pin(I) {
        direction : input;
        max_transition : 2100.0;
        capacitance : 0.002000;
        fanout_load : 1;
    }
    pin(Z) {
        direction : output;
        max_capacitance : 0.175000;
        max_fanout : 58;
        max_transition : 1400.0;
        function : "(I)'";
        timing() {
            related_pin      : "I";
            timing_sense    : negative_unate;
            ...
            ccsn_first_stage ( ) {
                is_needed      : true;
                is_inverting   : true;
                stage_type     : both;
                miller_cap_rise : 0.00055;
                miller_cap_fall : 0.00084;

                dc_current (ccsn_dc_29x29) {
                    index_1 ("-1.200, -0.600, -0.240, -0.120, 0.000, 0.060, 0.120, 0.180, \
                                0.240, 0.300, 0.360, 0.420, 0.480, 0.540, 0.600, 0.660, \
                                0.720, 0.780, 0.840, 0.900, 0.960, 1.020, 1.080, 1.140, \
                                1.200, 1.320, 1.440, 1.800, 2.400");
                    index_2 ("-1.200, -0.600, -0.240, -0.120, 0.000, 0.060, 0.120, 0.180, \
                                0.240, 0.300, 0.360, 0.420, 0.480, 0.540, 0.600, 0.660, \
                                0.720, 0.780, 0.840, 0.900, 0.960, 1.020, 1.080, 1.140, \
                                1.200, 1.320, 1.440, 1.800, 2.400");
                    values ("619.332000, 0.548416, 0.510134, 0.491965, 0.470368, \
                                ...

```

```

        -0.390604, -0.394495, -0.403571, -579.968000");
    }
    output_voltage_rise ( ) {
        vector (ccsn_timing_lut_5) {
            index_1(175.000);
            index_2(0.004000);
            index_3 ("104.222, 127.996, 144.729, 159.367, 176.983");
            values ("1.080, 0.840, 0.600, 0.360, 0.120");
        }
    ...
}

output_voltage_fall ( ) {
    vector (ccsn_timing_lut_5) {
        index_1(175.000);
        index_2(0.004000);
        index_3 ("104.222, 127.996, 144.729, 159.367, 176.983");
        values ("1.080, 0.840, 0.600, 0.360, 0.120");
    }
    ...
}

propagated_noise_low ( ) {
    vector (ccsn_prop_lut_5) {
        index_1(0.6000);
        index_2(1365.00);
        index_3(0.004000);
        index_4 ("640.90, 679.55, 711.76, 755.45, 793.68");
        values ("0.0553, 0.0884, 0.1105, 0.0884, 0.0553");
    }
    ...
    vector (ccsn_prop_lut_5) {
        index_1(0.6500);
        index_2(2730.00);
        index_3(0.019000);
        index_4 ("1298.73, 1379.15, 1484.78, 1599.75, 1687.38");
        values ("0.0927, 0.1483, 0.1854, 0.1483, 0.0927");
    }
}
}

propagated_noise_high ( ) {
    vector (ccsn_prop_lut_5) {
        index_1(0.6000);
        index_2(1365.00);
        index_3(0.004000);
        index_4 ("648.77, 688.99, 741.96, 793.08, 833.85");
        values ("1.0592, 0.9748, 0.9184, 0.9748, 1.0592");
    }
    ...
    vector (ccsn_prop_lut_5) {
        index_1(0.6500);
        index_2(2730.00);
        index_3(0.019000);
        index_4 ("1307.15, 1404.92, 1561.13, 1709.43, 1814.30");
        values ("1.0028, 0.8844, 0.8055, 0.8844, 1.0028");
    }
}
}

```

```

    } /* ccsn_first_stage */

} /* timing I -> Z */
} /* Z */
} /* cell(inv) */

} /* library *

```

Conditional Data Modeling in CCS Noise Models

The following attributes support conditional data modeling in pin-based CCS noise models:

- The `when` attribute in the `ccsn_first_stage` and `ccsn_last_stage` groups.
- The `mode` attribute.

The following syntax shows the `when` and `mode` support for pin-based CCS noise models:

```

cell(cell_name) {
    mode_definition (mode_name) {
        mode_value(namestring) {
            when : "Boolean expression";
            sdf_cond : "Boolean expression";
        } ...
    } ...
    pin(pin_name) {
        direction : input;
        /* The following syntax supports pin-based ccs noise */
        /* ccs noise first stage for Condition 1 */
        ccsn_first_stage() {
            is_needed :true | false;
            when : "Boolean expression";
            mode (mode_name, mode_value);
            ...
        }
        ...
        /* ccs noise first stage for Condition n */
        ccsn_first_stage() {
            is_needed :true | false;
            when : "Boolean expression";
            mode (mode_name, mode_value);
            ...
        }
        pin(pin_name) {
            direction : output;
            /* ccs noise last stage for Condition 1 */
            ccsn_last_stage() {
                is_needed : true | false;
                when : "Boolean expression";
                mode (mode_name, mode_value);
                ...
            }
        }
    }
}

```

```

...
/* ccs noise last stage for Condition n */
ccsn_last_stage() {
    is_needed : true | false;
    when : "Boolean expression";
    mode (mode_name, mode_value);
    ...
}
timing() {
    ...
/* following are arc-based ccs noise */
ccsn_first_stage() {
    is_needed : true | false;
    ...
}
...
ccsn_last_stage() {
    is_needed : true | false;
    ...
}
}
}
}

```

when Attribute

The `when` attribute is a conditional attribute that is supported in pin-based CCS noise models in the `ccsn_first_stage` and `ccsn_last_stage` groups.

mode Attribute

The pin-based mode attribute is provided in the ccsn_first_stage and ccsn_last_stage groups for conditional data modeling. If the mode attribute is specified, mode_name and mode value must be predefined in the mode definition group at the cell level.

Example

```

library (csm13os120_typ) {
  technology ( cmos ) ;
  delay_model : table_lookup;
  lu_table_template(ccsn_dc_29x29) {
    variable_1 : input_voltage;
    variable_2 : output_voltage;
  }
  cell(inv0d0) {
    area : 0.75;
    mode_definition(rw) {
      mode_value(read) {
        when : "I";
        sdf_cond : "I == 1";
      }
    }
  }
}

```

```

mode_value(write) {
    when : "!I";
    sdf_cond : "I == 0";
}
}
pin(I) {
    direction : input;
    max_transition : 2100.0;
    capacitance : 0.002000;
    fanout_load : 1;
    ...
}
pin(ZN) {
    direction : output;
    max_capacitance : 0.175000;
    max_fanout : 58;
    max_transition : 1400.0;
    function : "(I)";
/* pin-based CCS noise first stage for Condition 1 */
ccsn_first_stage () {
    ...
    when : "I"; /* or using mode as next commented line */
/* mode(rw, read); */
    dc_current (ccsn_dc_29x29) { ... }
    output_voltage_rise ( ) { ... }
    output_voltage_fall ( ) { ... }
    propagated_noise_low ( ) { ... }
    propagated_noise_high ( ) { ... }
} /* ccsn_last_stage */
/* pin-based CCS noise last stage for Condition 2 */
ccsn_last_stage () {
    ...
    when : "!I"; /* or using mode as next commented line */
/* mode(rw, read); */
    dc_current (ccsn_dc_29x29) { ... }
    output_voltage_rise ( ) { ... }
    output_voltage_fall ( ) { ... }
    propagated_noise_low ( ) { ... }
    propagated_noise_high ( ) { ... }
} /* ccsn_last_stage */

timing() {
    related_pin : "I";
    timing_sense : negative_unate;
    ...
} /* timing I -> Z */
} /* Z */
} /* cell(inv0d0) */
} /* library */

```

CCS Noise Modeling for Unbuffered Cells With a Pass Gate

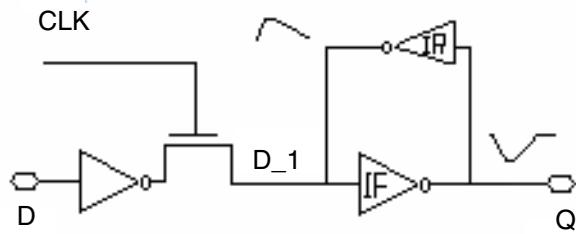
Unbuffered input and output latches are a special type of cell that has an internal memory node connected to an input or output pin. To increase the speed of the design and lower power consumption, these cells do not use inverters.

[Figure 16-1](#) and [Figure 16-2](#) show the schematics of a typical unbuffered output latch and an unbuffered input latch, respectively. The major difference between an unbuffered output cell and unbuffered input cell and a regular cell is as follows:

- Unbuffered Output Cell

An unbuffered output cell has the *feedback*, or back-driving path, from the unbuffered output pin to an internal node. In [Figure 16-1](#), Q is connected to internal node D_1 through the IR inverter.

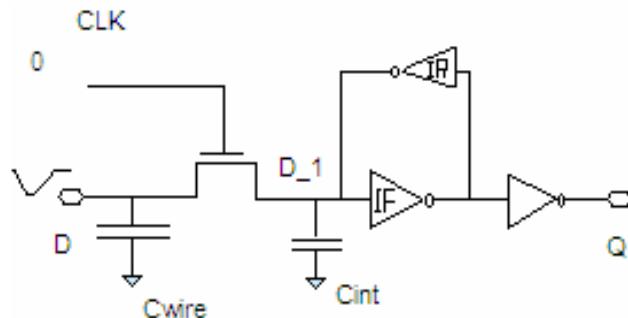
Figure 16-1 Unbuffered Output Latch



- Unbuffered Input Cell

The input pin of an unbuffered cell is not buffered and can be connected through a pass gate to the internal node. (A pass gate is a special gate that has an input and an output and a control input. If the control is set to true, the output is driven by the input. Otherwise, it is a floating output.) For example, in [Figure 16-2](#), D is connected to internal node D_1 through a pass gate.

Figure 16-2 Unbuffered Input Latch



To correctly model this category of cells in Liberty syntax, you must determine:

- If a pin is buffered or unbuffered.
 - If a pin is implemented with a pass gate.
 - If the `ccsn_*_stage` information models a pass gate.

Syntax for Unbuffered Output Latches

The following syntax supports unbuffered output latches.

Syntax

```

/* unbuffered output pin */
pin (pin_name) {
    direction : inout | output;
    is_unbuffered : true | false ;
    has_pass_gate : true | false ;
    ccsn_first_stage () {
        is_pass_gate : true | false;
        ...
    }
    ...
    ccsn_last_stage () {
        is_pass_gate : true | false;
        ...
    }
    ...
    timing() {
        ccsn_first_stage () {
            is_pass_gate : true | false;
            ...
        }
        ...
        ccsn_last_stage () {
            is_pass_gate : true | false;

```

```
    ...
}
...
}
}
pin (pin_name) {
    direction : input | inout;
    is_unbuffered : true | false ;
    has_pass_gate : true | false ;
    ccsn_first_stage () {
        is_pass_gate : true | false;
    ...
}
...
ccsn_last_stage () {
    is_pass_gate : true | false;
...
}
...
timing() {
    ccsn_first_stage () {
        is_pass_gate : true | false;
    ...
}
...
ccsn_last_stage () {
    is_pass_gate : true | false;
...
}
...
}
}
```

Pin-Level Attributes

The following attributes are pin-level attributes for unbuffered output latches.

is_unbuffered Attribute

The `is_unbuffered` simple Boolean attribute indicates that a pin is unbuffered. This optional attribute can be specified on the pins of any library cell. The default is `false`.

has_pass_gate Attribute

The `has_pass_gate` simple Boolean attribute can be defined in a pin group to indicate whether the pin is internally connected to at least one pass gate.

ccsn_first_stage Group

The `ccsn_first_stage` group specifies CCS noise for the first stage of the channel-connected block (CCB). When the `ccsn_first_stage` group is defined at the pin level, it can only be defined in an input pin or an inout pin.

The `ccsn_first_stage` group syntax models back-driving CCS noise propagation information from the output pin to the internal node.

is_pass_gate Attribute

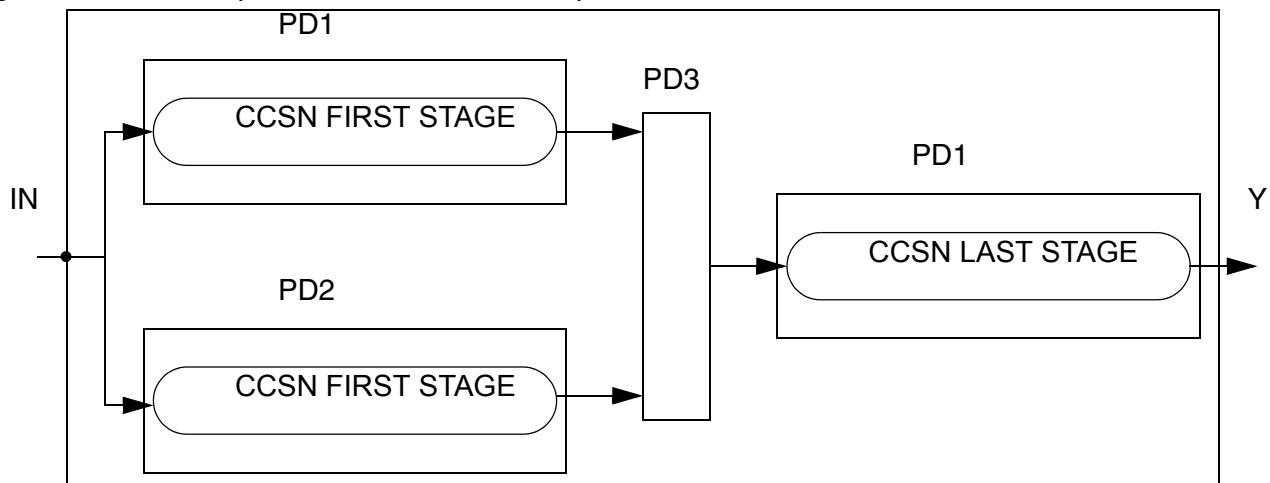
The `is_pass_gate` Boolean attribute is defined in a `ccsn_*_stage` group (such as `ccsn_first_stage`) to indicate whether the `ccsn_*_stage` information is modeled for a pass gate. The attribute is optional and its default is `false`.

CCS Noise Modeling for Multivoltage Designs

In multivoltage designs, a complex macro cell can have multiple power domains with different power supplies internal to the cell. The internal power supplies that provide power to some of the inputs and outputs of the channel-connected blocks (CCBs), cannot be modeled at the pin level. Therefore, pin-level attributes do not correctly describe the operation of the CCS noise stages for these channel-connected blocks. To correctly model the CCS noise stages, specify the internal power supplies in the `ccsn_first_stage` and `ccsn_last_stage` groups that are specified within the `pin` or `timing` groups.

[Figure 16-3](#) shows a macro cell with three power domains, PD1, PD2, and PD3. The input pin, IN, is connected to the channel-connected blocks (not shown in the figure) for two CCS noise stages in PD1 and PD2. The outputs of the channel-connected blocks go to PD3. The CCS noise stage for the output pin, Y, is in PD1 and is driven by an internal stage in PD3. Therefore, the example in [Figure 16-3](#) shows that the channel-connected block inputs or outputs for the CCS stages do not always map to a pin.

Figure 16-3 A Complex Macro Cell With Multiple Power Domains



To model such CCS noise stages, use the `output_signal_level` and the `input_signal_level` attributes respectively within the `ccsn_first_stage` and `ccsn_last_stage` groups, as shown in [Example 16-2](#).

Note:

If a required `input_signal_level` or `output_signal_level` attribute specification is missing from the `ccsn_first_stage` and `ccsn_last_stage` groups, the Library Compiler tool generates the LBDB-706 error message.

[Example 16-2](#) shows both pin and timing arc-based CCS noise models of low-to-high level-shifter cells.

Example 16-2 A Library Description of a Level-Shifter Cell Modeled for CCS Noise

```

library (test) {
    technology(cmos);
    nom_voltage : 1.080000;
    voltage_unit : "1V";
    voltage_map(VDD1,1.080000);
    voltage_map(GND1,0.000000);
    voltage_map(VDD2,0.900000);
    ...
    cell (LVL_SHIFT_L2H_1) {
        is_level_shifter : true;
        level_shifter_type : LH;
        input_voltage_range(0.900000,1.320000);
        output_voltage_range(0.900000,1.320000);
        pg_pin (VDD) {
            voltage_name : VDD1;
            ...
        }
        pg_in (VDDL) {
            voltage_name : VDD2;
            ...
        }
    }
}
  
```

```

}
pin (I) {
    direction : input;
    level_shifter_data_pin : true;
    related_ground_pin : VSS;
    related_power_pin : VDDL;
    ccsn_first_stage () /* pin-based model */
        is_needed : true;
        stage_type : both;
    ...
    output_signal_level : VDD1;
    /* Specifies that the power supply to the ccsn_first_stage output
       is VDD1. The power supply to the ccsn_first_stage input is
       specified at the pin level by the related_power_pin attribute
       as VDDL. */

    dc_current (ccsn_dc) ...
}
pin (Z) {
    direction : output;
    power_down_function : "!VDD + !VDDL + VSS";
    function : "I";
    related_ground_pin : VSS;
    related_power_pin : VDD;
    ...
    ccsn_last_stage () /* pin-based model */
        is_needed : "true";
        stage_type : "both";
    ...
    input_signal_level : VDD2;
    /* Specifies that the power supply to the ccsn_last_stage input
       is VDD2. The power supply to the ccsn_last_stage output is
       specified at the pin level by the related_power_pin attribute
       as VDD. */

    dc_current (ccsn_dc) ...
}
...
}
cell (LVL_SHIFT_L2H_2) {
    is_level_shifter : true;
    level_shifter_type : LH;
    input_voltage_range(0.900000,1.320000);
    output_voltage_range(0.900000,1.320000);
    pg_pin (VDD) {
        voltage_name : VDD1;
    ...
}
pg_in (VDDL) {
    voltage_name : VDD2;
    ...
}
pin (I) {

```

```

        direction : input;
        level_shifter_data_pin : true;
        related_ground_pin : VSS;
        related_power_pin : VDDL;
    }
    pin (Z) {
        direction : output;
        power_down_function : "!VDD + !VDDL + VSS";
        function : "I";
        related_ground_pin : VSS;
        related_power_pin : VDD;
        ...
        timing() {
            related_pin : I;
            ...
            ccsn_first_stage () /* arc-based model */
                is_needed : true;
                stage_type : both;
                ...
                output_signal_level : VDD1;
                dc_current (ccsn_dc)...
            }
            ccsn_last_stage () /* arc-based model */
                is_needed : "true";
                stage_type : "both";
                ...
                input_signal_level : VDD1;
                dc_current (ccsn_dc)...
            }
        }
        ...
    }
    ...
}

```

Referenced CCS Noise Modeling

For CCS noise characterization, a circuit is partitioned into channel-connected blocks (CCBs). An input pin might simultaneously drive multiple channel-connected blocks (CCBs).

For different input states and capacitive loads, you can represent each channel-connected block using multiple `input_ccb` and `output_ccb` groups. The `input_ccb` and `output_ccb` groups have names and are defined in `pin` groups.

In each timing arc, you can use these names to reference the `input_ccb` or `output_ccb` groups that

- Contribute to the noise but do not propagate the noise in the timing arc; this also applies to pin-level `receiver_capacitance` groups
- Propagate the noise in the timing arc

Because you define the `input_ccb` and `output_ccb` groups only in the `pin` groups, this model eliminates multiple definitions of the same channel-connected block noise in different timing groups, such as for a `ccsn_last_stage` group representing an inverter in a conventional two-stage CCS noise model.

To better model noise, the `input_ccb` and `output_ccb` groups include the `output_voltage_rise` and `output_voltage_fall` groups. The `output_voltage_rise` and `output_voltage_fall` values are characterized with driver waveforms instead of ramp waveforms as in the conventional two-stage CCS noise model.

For the tool to compile the referenced CCS noise model, set the following variable to `true` before running the `read_lib` command:

```
lc_shell> set lc_enhanced_ccs true
```

For more information about the conventional two-stage CCS noise model, see “[CCS Signal Integrity Modeling Syntax](#)” on page 16-2.

Modeling Syntax

```
library (library_name) {
    cell (cell_name) {
        driver_waveform : "waveform_name";
        driver_waveform_rise : "waveform_name";
        driver_waveform_fall "waveform_name";
        ...
        pin (pin_name) {
            direction : input;
            ...
            input_ccb (input_ccb_name1) {
                when : logical_expression;
                mode(mode_name, mode_value);
                related_ccb_node : "spice_node_name1";
                output_voltage_rise {
                    ...
                }
                ...
            }
            input_ccb (input_ccb_name2) {
                when : logical_expression
                mode(mode_name, mode_value);
            }
        }
    }
}
```

```

related_ccb_node : "spice_node_name2";
output_voltage_rise {
    ...
}
...
}
...
receiver_capacitance (lu_template) {
    when : logical_expression;
    mode(mode_name, mode_value);
    active_input_ccb(input_ccb_name1, input_ccb_name2,...]);
} /* end receiver_capacitance group */
} /* end pin a group */
pin (pin_name) {
    direction : output;
    ...
    output_ccb(output_ccb_name1) {
        when : logical_expression;
        mode(mode_name, mode_value);
        related_ccb_node : "spice_node_name3";
    }
    timing() /* Arc has propagating noise (full arc-based) */
    ...
    active_input_ccb(input_ccb_name1, input_ccb_name2, ...);
    propagating_ccb(input_ccb_name2, output_ccb_name);
}
timing() /* Arc has no propagating noise because it involves
            three inverting stages*/
    ...
    active_input_ccb(input_ccb_name1, input_ccb_name2);
    active_output_ccb(output_ccb_name);
}
} /* end pin group */
} /* end cell group */
} /* end library group */

```

Cell-Level Attributes

This section describes the optional cell-level attributes specific to the referenced CCS noise model.

driver_waveform Attribute

The `driver_waveform` attribute specifies a cell-specific driver waveform. The specified `driver waveform` must be a predefined `driver_waveform_name` attribute value in the **library-level normalized_driver_waveform group table**.

If you do not specify this attribute, the tool uses a default driver waveform from the `normalized_driver_waveform table`.

driver_waveform_rise and driver_waveform_fall Attributes

The `driver_waveform_rise` and `driver_waveform_fall` specify the rise and fall driver waveforms. These attributes override the `driver_waveform` attribute.

For more information about driver waveform modeling syntax, see “[Driver Waveform Support](#)” on page [11-90](#).

Pin-Level Groups

This section describes the pin-level groups and attributes specific to the referenced CCS noise model.

input_ccb and output_ccb Groups

The `input_ccb` and `output_ccb` groups include all the attributes and subgroups of the `ccsn_first_stage` and `ccsn_last_stage` groups respectively.

Except for pins where the `input_ccb` and `output_ccb` groups are not required, you must name all these groups so that they can be referenced. For the exceptions, you must set the `is_needed` attribute to `false` in the `input_ccb` and `output_ccb` groups.

The name includes the pin identifier and the group identifier. The pin identifier is the same in all the `input_ccb` and `output_ccb` groups of the pin. The pin identifier enables homogeneous treatment of these groups in buses and bundles.

For example, in the `input_ccb("CCB_D1"){...}` group, the `CCB_D` part of the name indicates that the channel-connected block group belongs to pin D and `1` indicates that this is the first `input_ccb` group of pin D.

Note:

The library characterization tools can use an additional cell identifier in the `input_ccb` and `output_ccb` group names.

Conditional Data Modeling

Use the `mode` and `when` attributes in the `input_ccb` and `output_ccb` groups for conditional data modeling.

related_ccb_node Attribute

The optional `related_ccb_node` attribute specifies the SPICE node in the subcircuit netlist that is used for the `dc_current` table characterization. The attribute is defined in the `input_ccb` group of an input pin and the `output_ccb` group of an output pin.

output_voltage_rise and output_voltage_fall Groups

The `output_voltage_rise` and `output_voltage_fall` groups have the following differences from those in the CCS noise stage groups.

- The output waveform specified by the `index_3` and `values` attributes is characterized using a driver waveform and not a ramp waveform. So, the `index_1` or `input_net_transition` is a slew value based on slew trip points and library slew derate and not a rail-to-rail ramp transition time.
- The `index_1` (`input_net_transition`) values must match one of the `index_1` values of the driver waveform specified at the cell-level.
- The `index_2` (`total_output_net_capacitance`) values must match one of the `index_2` values of the `cell_rise` and `cell_fall` groups.
- The values at `index_1` and `index_2` of all the `output_voltage_rise` or `output_voltage_fall` groups in an `input_ccb` or `output_ccb` group must form a grid without duplicated or missing points.

For an `input_ccb` group that does not directly drive an output pin, the number of `output_voltage_rise` groups must be $M \times 1$. For an `output_ccb` or an `input_ccb` that directly drives an output pin, the number of groups must be $M \times N$.

M is the number of `index_1` values and N is the number of `index_2` values.

Note:

For libraries with 7×7 or 8×8 nonlinear delay model (NLDM) tables, both M and N can be 4.

Timing-Level Attributes

This section describes the timing-level attributes specific to the referenced CCS noise model.

active_input_ccb Attribute

The `active_input_ccb` attribute lists the `input_ccb` groups of the input pin that are active or switching in the timing arc or the receiver capacitance load but do not propagate the noise to the output pin. In the timing group, the input pin is specified by the `related_pin` attribute.

You can also specify this attribute in the `receiver_capacitance` group of the input pin.

active_output_ccb Attribute

The `active_output_ccb` attribute lists the `output_ccb` groups in the timing arc that drive the output pin, but do not propagate the noise. You must define both the `output_ccb` and timing groups in the same pin group.

propagating_ccb Attribute

The `propagating_ccb` attribute lists all the channel-connected block groups that propagate the noise in a particular timing arc, to the output pin.

In the list, the first name is the `input_ccb` group of the input pin (specified by the `related_pin` attribute in the timing group). The second name, if present, is for the `output_ccb` group of the output pin.

Note:

You can list at most two names. This attribute does not support timing arcs with three or more inverting stages. For such cases and complex circuits with converging paths, reference the `input_ccb` group name in the `active_input_ccb` attribute and the `output_ccb` group name in the `active_output_ccb` attribute.

Examples

The following example shows a simple noise model with two timing arcs sharing the same output channel-connected block.

```
cell (AND2) {
  ...
  pin (A) {
    ...
    input_ccb("CCB_A") {
      related_ccb_node : "net1:15";
      ...
    }
  }
  pin (B) {
    ...
    input_ccb("CCB_B") {
      related_ccb_node : "net1:15";
      ...
    }
    input_ccb("CCB_CP2") {
      related_ccb_node : "net3:3";
      ...
    }
  }
  pin (Y) {
    ...
    output_ccb("CCB_Y") {
```

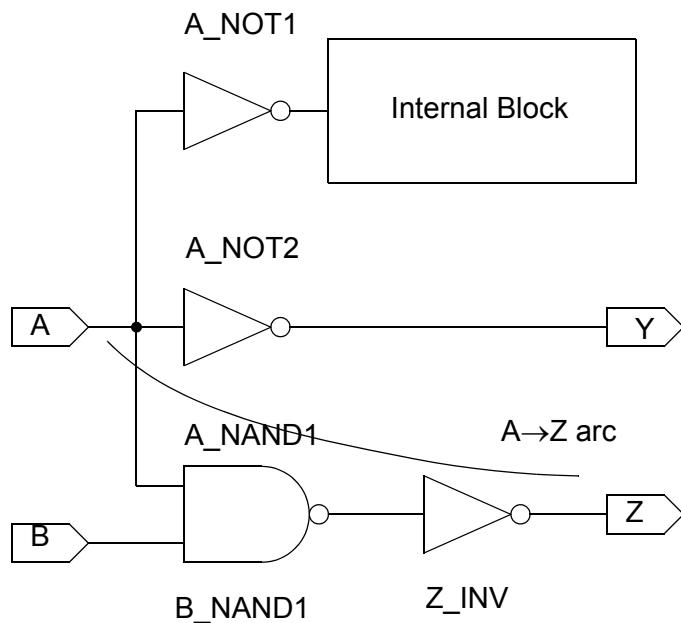
```

related_ccb_node : "net1:15";
...
}
timing () {
    related_pin : A;
    propagating_ccb("CCB_A", "CCB_Y");
}
timing () {
    related_pin : B;
    propagating_ccb("CCB_B", "CCB_Y");
}
...
}
}

```

The following figure is a schematic with the input, A, driving three logic gates. The relevant channel-connected blocks are shown.

Figure 16-4 A Schematic With Multiple Active Channel-Connected Blocks



The following table shows which of the four active channel-connected blocks propagate the noise in the A→Z timing arc.

Active channel-connected block	Propagates noise in A→Z arc?
A_NOT1	No

Active channel-connected block	Propagates noise in A→Z arc?
---------------------------------------	-------------------------------------

A_NOT2	No
A_NAND1	Yes
Z_INV	Yes

The following example shows how the blocks are referenced in the timing group.

```
pin(Z) {
    direction : output;
    timing () { /* A→Z arc */
        related_pin : A;
        ...
        active_input_ccb("A_NOT1", "A_NOT2");
        propagating_ccb("A_NAND1", "Z_INV");
    } /* end timing */
}
```

17

Composite Current Source Power Modeling

This chapter provides an overview of composite current source (CCS) modeling to support advanced technologies. It covers the syntax for CCS power modeling in the following sections:

- [Composite Current Source Power Modeling](#)
- [Compact CCS Power Modeling](#)
- [Composite Current Source Dynamic Power Examples](#)

Composite Current Source Power Modeling

The library nonlinear power model format captures leakage power numbers in multiple input combinations to generate a state-dependent table. It also captures dynamic power of various input transition times and output load capacitance to create the state-dependent and path-dependent internal energy data.

The composite current source (CCS) power modeling format extends current library models to include current-based waveform data to provide a complete solution that addresses static and dynamic power. It also addresses dynamic IR drop. The following are features of this approach as compared to the nonlinear power model:

- Creates a single unified power library format suitable for power optimization, power analysis, and rail analysis.
 - Captures a supply current waveform for each power or ground pin.
 - Provides finer time resolution.
 - Offers full multivoltage support.
 - Captures equivalent parasitic data to perform fast and accurate rail analysis.
 - Reduces the characterization runtime.
-

Cell Leakage Current

Because CCS power is current-based data, leakage current on the power and ground pins is captured instead of leakage power as specified in the nonlinear power model format. For information about gate leakage, see “[gate_leakage Group](#)” on page 17-4. The leakage current syntax is as follows:

Example 17-1 Leakage Current Syntax

```
cell(cell_name) {
    ...
    leakage_current() {
        when : "Boolean expression";
        pg_current(pg_pin_name) {
            value : float;
        }
        ...
    }
    leakage_current() { /* without the when statement */
        /* default state */
        ...
    }
}
```

Current conservation means that the sum of all current values must be zero. A positive value means power pin current, and a negative value means ground pin current.

If you have two power and ground pins in your design, and you have already specified the power current value to 2.0, you do not have to specify the ground current value, because the tool infers that it must be -2.0 based on current conservation.

For multiple power and ground pins, you must use the regular format because it provides pg_current, which allows you to specify the power and ground names. For example, if you have two power pins, you must specify the value for each pin.

Again, a simplified format is allowed for a cell with a single power and ground pin. For this case, no pg_current group is required within a leakage_current group.

Example 17-2 Leakage Current Format Simplified

```
cell(cell_name) {
    ...
    leakage_current() /* without pg_current group */
        when : "Boolean expression";
        value : float;
    }

    leakage_current() /* without the when statement */
        /* default state */
        ...
}
```

Checking Leakage Current Syntax

The Library Compiler tool automatically checks the syntax for leakage current and reports any problems it encounters. For information about the types of checks the Library Compiler tool performs for leakage current, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Gate Leakage Modeling in Leakage Current

The syntax for these power models is described in the following section.

Syntax

```
cell(cell_name) {
    ...
    leakage_current() {
        when : "Boolean expression";
        pg_current(pg pin name) {
            value : float;
    }
}
```

```

...
gate_leakage(input pin name) {
    input_low_value : float;
    input_high_value : float;
}
...
}
...
leakage_current() {
/* group without when statement */
/* default state */
...
}
}

```

gate_leakage Group

This group specifies the cell's gate leakage current on input or inout pins within the `leakage_current` group in a cell. For information about cell leakage, see [“Cell Leakage Current” on page 17-2](#).

The following information pertains to a `gate_leakage` group:

- Groups can be placed in any order if there are more than one `gate_leakage` groups within a `leakage_current` group.
- Leakage current of a cell is characterized with opened outputs, which means outputs of a modeling cell do not drive any other cells. Outputs are assumed to have zero static current during the measurement.
- A missing `gate_leakage` group is allowed for certain pins.
- Current conservation is applicable if it can be applied to higher error tolerance.

input_low_value Attribute

This attribute specifies gate leakage current on an input or inout pin when the pin is in a `low` state.

- A negative float value is required.
- The gate leakage current is measured from the power pin of the cell to the ground pin of its driver cell.
- The input pin is pulled low.
- The `input_low_value` attribute is not required for a `gate_leakage` group.
- Defaults to 0 if no `gate_leakage` group is specified for certain pins.
- Defaults to 0 if no `input_low_value` attribute is specified in the `gate_leakage` group.

input_high_value Attribute

This attribute specifies gate leakage current on an input or inout pin when the pin is in a `high` state.

- A positive float value is required.
- The gate leakage current is measured from the power pin of its driver cell to the ground pin of the cell.
- The input pin is pulled high.
- The `input_high_value` is not required for a `gate_leakage` group.
- Defaults to 0 if no `gate_leakage` groups is specified for certain pins.
- Defaults to 0 if no `input_high_value` is specified in the `gate_leakage` group.

Checking Gate Leakage Current Syntax

The Library Compiler tool automatically checks the conditions under a `leakage_current` group and reports any problems it encounters. For information about the types of checks the Library Compiler tool performs for leakage current, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Intrinsic Parasitic Models

You can use the syntax in [Example 17-3](#) for intrinsic parasitic models. The syntax consists of two parts: one is intrinsic resistance and the other is intrinsic capacitance.

Example 17-3 Intrinsic Parasitic Model

```
cell (cell_name) {
    mode_definition (mode_name) {
        mode_value(namestring) {
            when : "Boolean expression";
            sdf_cond : "Boolean expression";
        }
    ...
    intrinsic_parasitic() {
        mode (mode_name, mode_value);
        when : "Boolean expression";
        intrinsic_resistance(pg_pin_name) {
            related_output : output_pin_name;
            value : float;
        }
        intrinsic_capacitance(pg_pin_name) {
            value : float;
        }
    }
}
```

```

        }

intrinsic_parasitic() {
    /*without when statement */
    /* default state */

}
}

```

[Example 17-4](#) shows a typical intrinsic parasitic model of a cell.

Example 17-4 Conditional Data Modeling for Intrinsic Parasitic Model By Mode

```

library (csm13os120_typ) {
    technology ( cmos ) ;
    delay_model : table_lookup;
    lu_table_template(ccsn_dc_29x29) {
        variable_1 : input_voltage;
        variable_2 : output_voltage;
    }
    cell(inv0d0) {
        area : 0.75;
        pg_pin(V1) {
            voltage_name : VDD1;
            pg_type : primary_power;
        }
        pg_pin(G1) {
            voltage_name : GND1;
            pg_type : primary_ground;
        }
        mode_definition(rw) {
            mode_value(read) {
                when : "A1";
                sdf_cond : "A1 == 1";
            }
            mode_value(write) {
                when : "!A1";
                sdf_cond : "A1 == 0";
            }
        }
        pin(A1) {
            direction : input;
            capacitance : 0.1 ;
            related_power_pin : V1;
            related_ground_pin : G1;
        }
        pin(A2) {
            direction : input;
            capacitance : 0.1 ;
            related_power_pin : V1;
            related_ground_pin : G1;
        }
        pin(ZN) {

```

```

direction : output;
max_capacitance : 0.1;
function : "!A1+A2";
related_power_pin : V1;
related_ground_pin : G1;
timing() {
    timing_sense : "negative_unate"
    related_pin : "A1";
    ...
}
pin(ZN1) {
    direction : output;
    max_capacitance : 0.1;
    function : "!A1";
    related_power_pin : V1;
    related_ground_pin : G1;
    timing() {
        timing_sense : "negative_unate"
        related_pin : "A1 A2";
    ...
}
}
intrinsic_parasitic() {
    mode(rw, read);
    intrinsic_resistance(G1) {
        related_output : "ZN";
        value : 9.0;
    }
    intrinsic_capacitance(G1) {
        value : 8.2;
    }
}
} /* cell(inv0d0) */
} /* library

```

Voltage-Dependent Intrinsic Parasitic Models

Intrinsic parasitics are conventionally modeled as voltage-independent or steady-state values. However, intrinsic parasitics are voltage-dependent. To better represent intrinsic parasitics in a CCS power model, use a lookup table for intrinsic parasitics instead of a single steady-state value. You can use the steady-state value when your design requirements are not critical.

The lookup table is one-dimensional and consists of intrinsic parasitic values for different values of VDD. You can selectively add these values to any `intrinsic_resistance` or `intrinsic_capacitance` subgroup. Use lookup tables when correct estimation of voltage drops is critical, such as power-switch designs.

The following are the advantages of using lookup tables.

- Accurate estimation of peak-inrush current and wake-up time
- Optimal power-up and power-down sequencing
- Optimal power-switch design, that is, minimum number of used and placed power-switch cells

[Example 17-5](#) shows the syntax for a voltage-dependent intrinsic parasitic model.

[Example 17-6](#) shows a typical voltage-dependent intrinsic parasitic model.

Example 17-5 Syntax for Intrinsic Parasitic Model With Lookup Tables

```
lu_table_template (template_name) {
variable_1 : pg_voltage | pg_voltage_difference ;
index_1 ("float, ... float");
}
cell (cell_name) {
...
    intrinsic_parasitic() {
        when : "Boolean expression";
        intrinsic_resistance (pg_pin_name) {
            related_output : output_pin_name;
            value : float;
            reference_pg_pin : pg_pin_name;
            lut_values (template_name) {
                index_1 ("float, ... float");
                values ("float, ... float");
            }
        }
        intrinsic_capacitance(pg_pin_name) {
            value : float;
            reference_pg_pin : pg_pin_name;
            lut_values (template_name) {
                index_1 ("float, ... float");
                values ("float, ... float");
            }
        }
    }
}
...
}
```

Example 17-6 Voltage-Dependent Intrinsic Parasitic Model Using Lookup Tables

```
library(example_library) {
    .....
    lu_table_template ( test_voltage ) {
        variable_1 : pg_voltage;
        index_1 ( "0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0" );
    }
    cell (AND3) {
        .....
    }
}
```

```

intrinsic_parasitic() {
    when : "A1 & A2 & ZN";
    intrinsic_resistance(G1) {
        related_output : "ZN";
        value : 9.0;
        reference_pg_pin : G1;
        lut_values ( test_voltage ) {
            index_1 ( "0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1.0" );
            values ( "0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1.0" );
        }
    }
    intrinsic_capacitance(G2) {
        value : 8.2;
        reference_pg_pin : G2;
        lut_values ( test_voltage ) {
            index_1 ( "0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1.0" );
            values ( "0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1.0" );
        }
    }
    intrinsic_resistance(G1) {
        related_output : "ZN1";
        value : 62.2;
    }
}
intrinsic_parasitic() {
/* default state */
    intrinsic_resistance(G1) {
        related_output : "ZN";
        value : 9.0;
        reference_pg_pin : G1;
        lut_values ( test_voltage ) {
            index_1 ( "0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1.0" );
            values ( "0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1.0" );
        }
    }
    intrinsic_resistance(G1) {
        related_output : "ZN1";
        value : 9.0;
    }
    intrinsic_capacitance(G2) {
        value : 8.2;
        reference_pg_pin : G2;
        lut_values ( test_voltage ) {
            index_1 ( "0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1.0" );
            values ( "0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1.0" );
        }
    }
}

```

```

        }
    }
    .....
} /* end of cell AND3 */
}

```

Library-Level Group

The following library-level group models voltage-dependent intrinsic parasitics.

lu_table_template Group

This group defines the template for the `lut_values` group. The `lu_table_template` group includes a one-dimensional variable, `variable_1`. The valid values of the `variable_1` variable are `pg_voltage` and `pg_voltage_difference`. When the `variable_1` variable is set to the `pg_voltage` value, the values of the intrinsic capacitance or resistance directly vary with the power supply voltage of the cell. When the `variable_1` variable is set to the `pg_voltage_difference` value, the values of the intrinsic capacitance or resistance vary with the difference between the power supply voltage and the power pin voltage specified by the `reference_pg_pin` attribute.

Note:

The `reference_pg_pin` attribute specifies the reference pin for the `intrinsic_resistance` and `intrinsic_capacitance` groups. The reference pin must be a valid PG pin.

Pin-Level Group

The following pin-level group models voltage-dependency in intrinsic parasitics by using lookup tables.

lut_values Group

To use the lookup table for intrinsic parasitics, use the `lut_values` group. You can add the `lut_values` group to both the `intrinsic_resistance` and `intrinsic_capacitance` groups. The `lut_values` group uses the `variable_1` variable, which is defined within the `lu_table_template` group, at the library level.

Checking Intrinsic Parasitic Syntax

The Library Compiler tool automatically checks intrinsic parasitic model syntax for both steady-state and lookup table values, and reports any problems it encounters. For information about the types of the Library Compiler tool checks on the intrinsic parasitic model syntax, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Parasitics Modeling in Macro Cells

For macro cells, the `total_capacitance` group is provided within the `intrinsic_parasitic` group.

total_capacitance Group

The `total_capacitance` group specifies the macro cell's total capacitance on a power or ground net within the `intrinsic_parasitic` group.

- This group can be placed in any order if there is more than one `total_capacitance` group within an `intrinsic_parasitic` group.
- If the `total_capacitance` group is not defined for a certain power and ground pin, the value of capacitance defaults to 0.0. The default is provided by the tool.
- The parasitics modeling of the total capacitance in macros cells is not state dependent. This means that there is no state condition specified in the `intrinsic_parasitic` group.

Parasitics Modeling Syntax

```
cell (cell_name) {
    power_cell_type : enum(stdcell, macro);
    ...
    intrinsic_parasitic() {
        total_capacitance(pg pin name) {
            value : float;
        }
        ...
    }
    ...
}
```

Checking Parasitic Model Syntax

The Library Compiler tool automatically checks parasitic model syntax and reports any problems it encounters. For information about the types of checks the Library Compiler tool performs for parasitic model syntax, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Dynamic Power

Because CCS power is current-based data, instantaneous power data on the power or ground pin is captured instead of internal energy specified in the nonlinear power model format. The current-based data provides higher accuracy than the existing model.

In the CCS modeling format, instantaneous power data is specified as a table of current waveforms. The table is dependent on the transition time of a toggling input and the capacitance of the toggling outputs.

As the number of output pins increases in a cell, the number of waveform tables becomes large. However, the cell with multiple output pins (more than one output) does not need to be characterized for all possible output load combinations. Therefore, two types of methods can be introduced to simplify the captured data.

- Cross type - Only one output capacitance is swept, while all other output capacitances are held in a typical value or fixed value.
- Diagonal type - The capacitance to all the output pins is swept together by an identical value.

A table that is modeled based on these two types is defined as a sparse table. Otherwise it is defined as a dense table, meaning that all combinations of the output load variable are specified in tables.

Dynamic Power and Ground Current Table Syntax

You can use the following syntax for dynamic current:

```
pg_current_template(template_name_1) {
    variable_1 : input_net_transition;
    variable_2 : total_output_net_capacitance;
    variable_3 : time;
    index_1(float, );                  /* optional */
    index_2(float, );                  /* optional */
    index_3(float, );                  /* optional */
}

pg_current_template(template_name_2) {
    variable_1 : input_net_transition;
    variable_2 : total_output_net_capacitance;
    variable_3 : total_output_net_capacitance;
    variable_4 : time;
    index_1(float, );                  /* optional */
    index_2(float, );                  /* optional */
    index_3(float, );                  /* optional */
    index_4(float, );                  /* optional */
}
```

Dynamic Power Modeling in Macro Cells

The extensions to CCS dynamic power format provides more accurate models for macro cells. The current dynamic power model only supports current waveforms for single-input events.

The model can also be applied to memory modeling with synchronous events, which are triggered by toggling either a single `read_enable` or `write_enable`.

However, for asynchronous event, the read access can be triggered by more than one bit of the address bus toggling. To support asynchronous memory access for macro cells, use `min_input_switching_count` and `max_input_switching_count`, in dynamic power as shown in the next section.

The following syntax for dynamic power format provides more accurate models for macro cells:

```
...
cell(cell_name) {
    mode_definition(mode_name) {
        mode_value(namestring) {
            when : "Boolean expression";
            sdf_cond : "Boolean expression";
        }
    ...
    power_cell_type : enum(stdcell, macro)
    dynamic_current() {
        mode(mode_name, mode_value);
        when : "Boolean expression";
        related_inputs : "input_pin_name";
        switching_group() {
            min_input_switching_count : integer;
            max_input_switching_count : integer;
            pg_current(pg_pin_name) {
                vector(template_name) {
                    reference_time : float;
                    index_1(float);
                    index_2("float,...");
                    values("float,...");
                } /* end vector group */
            ...
        } /* end pg_current group */
        ...
    } /* end switching_group */
    ...
} /* end dynamic_current group */
...
} /* end cell group*/
...
```

The `min_input_switching_count` and `max_input_switching_count` attributes specify the number of bits in the input bus that are switching simultaneously while an asynchronous event occurs. A single switching bit can be defined by setting the same value in both attributes.

The following example shows that any three bits specified in `related_inputs` are switching simultaneously.

```
...
min_input_switching_count : 3;
max_input_switching_count : 3;
...
```

A range of switching bits can be defined by setting the minimum and maximum value. The following example shows that any 2, 3, 4 or 5 bits specified in `related_inputs` are switching simultaneously.

```
...
min_input_switching_count : 2;
max_input_switching_count : 5;
...
```

min_input_switching_count Attribute

This attribute specifies the minimum number of bits in an input bus that are switching simultaneously.

- The count must be integer.
- The count must greater than 0 and less than `max_input_switching_count`.

max_input_switching_count Attribute

This attribute specifies the maximum number of bits in an input bus that are switching simultaneously.

- The count must be integer.
- The count must greater than `min_input_switching_count`.
- The count must be less than the total number of bits listed in `related_inputs`.

Checking Dynamic Power Model Syntax

The Library Compiler tool automatically checks dynamic power model syntax and reports any problems it encounters. For information about the types of checks the Library Compiler tool performs for dynamic power models, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Examples for CCS Dynamic Power for Macro Cells

```
...
pg_current_template ( CCS_power_1 ) {
    variable_1 : input_net_transition ;
    variable_2 : time;
```

```

    }
    type(bus3) {
        base_type : array;
        bit_width : 3;
    ...
    }
...
cell ( example ) {
    bus(addr_in) {
        bus_type: bus3;
        direction : input;
    ...
}
    pin(data_in) {
        direction : input;
    ...
}
    ...
power_cell_type : macro;
dynamic_current() {
    when: "!WE";
    related_inputs : "addr_in";
    switching_group ( ) {
        min_input_switching_count : 1;
        max_input_switching_count : 3;
        pg_current (VSS) {
            vector ( CCS_power_1 ) {
                reference_time : 0.01;
                index_1 ( "0.01" )
                index_2 ( "4.6, 5.9, 6.2, 7.3" )
                values ( "0.002, 0.009, 0.134, 0.546" )
            }
            ...
            vector ( CCS_power_1 ) {
                reference_time : 0.01;
                index_1 ( "0.03" )
                index_2 ( "2.4, 2.6, 2.9, 4.0" )
                values ( "0.012, 0.109, 0.534, 0.746" )
            }
            vector ( CCS_power_1 ) {
                reference_time : 0.01;
                index_1 ( "0.08" )
                index_2 ( "1.0, 1.6, 1.8, 1.9" )
                values ( "0.102, 0.209, 0.474, 0.992" )
            }
            ...
        } /* pg_current */
    ...
} /* switching_group */
...
} /* dynamic_current */
...
...

```

```

intrinsic_parasitic() {
    total_capacitance(VDD) {
        value : 0.2;
    }
    ...
} /* intrinsic_parasitic */
...
...
leakage_current() {
    when : WE;
    gate_leakage(data_in) {
        input_low_value : -0.3;
        input_high_value : 0.5;
    }
    ...
} /* leakage_current */
...
} /* end of cell */
...

```

Conditional Data Modeling for Dynamic Current Example

```

library (csm13os120_typ) {
    technology ( cmos ) ;
    delay_model : table_lookup;
    lu_table_template(ccsn_dc_29x29) {
        variable_1 : input_voltage;
        variable_2 : output_voltage;
    }
    cell(inv0d0) {
        area : 0.75;
        pg_pin(V1) {
            voltage_name : VDD1;
            pg_type : primary_power;
        }
        pg_pin(G1) {
            voltage_name : GND1;
            pg_type : primary_ground;
        }
        mode_definition(rw) {
            mode_value(read) {
                when : "A1";
                sdf_cond : "A1 == 1";
            }
            mode_value(write) {
                when : "!A1";
                sdf_cond : "A1 == 0";
            }
        }
        power_cell_type : stdcell;
        dynamic_current() { /* dense table */
            mode(rw, read);
        }
    }
}

```

```

related_inputs : "A2";
related_outputs : "ZN ZN1";
switching_group() {
    output_switching_condition(rise rise);
    pg_current(V1) {
        vector(test_1) {
            reference_time : 23.7;
            index_1("0.8");
            index_2("0.7");
            index_3("10.4");
            index_4("8.2 8.5 9.1 9.4 9.8");
            values("0.7 34.6 3.78 92.4 100.1");
        }
        ...
    }
}
pin(A1) {
    direction : input;
    capacitance : 0.1 ;
    related_power_pin : V1;
    related_ground_pin : G1;
}
pin(A2) {
    direction : input;
    capacitance : 0.1 ;
    related_power_pin : V1;
    related_ground_pin : G1;
}
pin(ZN) {
    direction : output;
    max_capacitance : 0.1;
    function : "!A1+A2";
    related_power_pin : V1;
    related_ground_pin : G1;
    timing() {
        timing_sense : "negative_unate"
        related_pin : "A1";
        ...
    }
}
pin(ZN1) {
    direction : output;
    max_capacitance : 0.1;
    function : "!A1";
    related_power_pin : V1;
    related_ground_pin : G1;
    timing() {
        timing_sense : "negative_unate"
        related_pin : "A1 A2";
        ...
    }
}

```

```

    } /* cell(inv0d0) */
} /* library

```

Checking Power and Ground Current Syntax

The Library Compiler tool automatically checks power and ground current template syntax and reports any problems it encounters. For information about the types of checks the Library Compiler tool performs for power and ground current syntax, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Dynamic Current Syntax

The syntax in [Example 17-7](#) is used for instantaneous power data, which is captured at the cell level.

Example 17-7 Dynamic Current Syntax

```

cell(cell_name) {
    power_cell_type : enum(stdcell, macro)
    dynamic_current() {
        when : "Boolean expression";
        related_inputs : input_pin_name;
        related_outputs : output_pin_name;
        typical_capacitances(float, );/* applied for cross type;*/
        switching_group() {
            input_switching_condition(enum(rise, fall));
            output_switching_condition(enum(rise, fall));
            pg_current(pg_pin_name) {
                vector(template_name) {
                    reference_time : float;
                    index_output : output_pin_name; /* applied for
cross type;*/
                    index_1(float);
                    index_n(float);
                    index_n+1(float, );
                    values(float, );
                } /* vector */
            } /* pg_current */
        } /* switching_group */
    } /* dynamic_current */
} /* cell */

```

Note:

For the `input_switching_condition` and `input_switching_condition` groups, the `rise` and `fall` values can be separated by commas or space. In either case, the Library Compiler tool compiles them.

Checking Dynamic Current Syntax

The Library Compiler tool automatically checks dynamic current syntax and reports any problems it encounters. For information about the types of checks the Library Compiler tool performs for dynamic current syntax, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

Compact CCS Power Modeling

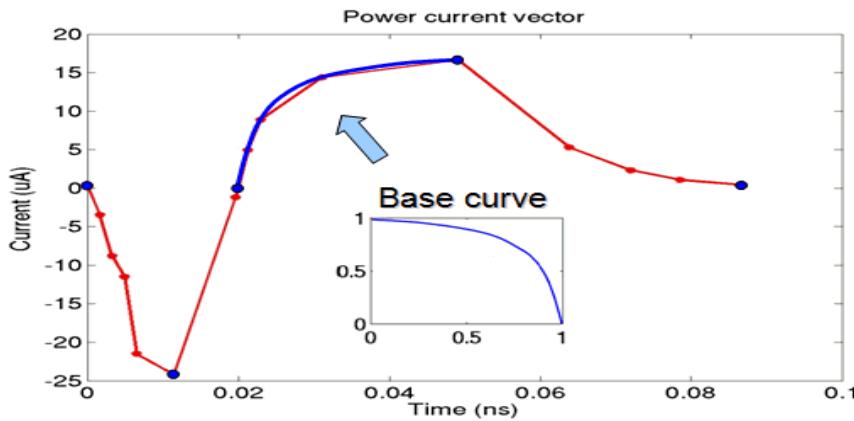
CCS power compaction uses base curve technology to significantly reduce the library size of CCS power libraries. Greater control of the Liberty file size allows you to include additional data points and more accurately capture CCS power data for dynamic current waveforms.

Base curve technology was first introduced for compact CCS timing. Each timing current waveform is split into two segments in the I-V domain, and the shape of each segment is modeled by a base curve that has a similar shape. This segmentation prevents direct modeling with the piecewise linear data points.

Compact CCS power modeling differs from compact CCS timing modeling in that power waveforms can include both positive sections and negative sections. They must be modeled in an $I(t)$ domain. In addition, the segmentation is more flexible. This is important because the current waveform shape might contain one or more bumps. The compact CCS power modeling segmentation points can be selected at:

- The point where the current waveform crosses zero
- The peak of a current bump

In [Figure 17-1](#), the red curve shows an example CCS power waveform in piecewise linear format with fifteen data points. Thirty float numbers must be stored in the library. This is an expensive storage cost for a single $I(t)$ waveform because libraries generally include a large number of $I(t)$ waveforms. In addition, the waveform shape is not smooth, despite the fifteen data points, due to the inefficiency of the piecewise linear representation. The current value error is larger than 5% in some regions of the waveform.

Figure 17-1 Power Current Vector

Segmenting the waveform and using base curve technology for each segment provides greater accuracy. In [Figure 17-1](#), the blue dots and blue curve show the waveform using base curve technology. The blue dots represent five segmentation points that divide the waveform into four sections. You can save the segmentation points as characteristic points and model the shape of each segment using a base curve. The blue curve is the third segment that can be represented by a base curve.

The following example describes the format for each current waveform:

$t_{start}, I_{start}, bcid_1, t_{ip1}, I_{ip1}, bcid_2, [t_{ip2}, I_{ip2}, bcid_3, \dots,] t_{end}, I_{end};$

The arguments are defined as follows:

t_{start}

The current start time.

I_{start}

The initial current.

t_{ip}

The time of an internal segmentation point.

I_{ip}

The current value of an internal segmentation point.

t_{end}

The time when transition ends and current value becomes stable.

I_{end}

The current value at the endpoint.

bcid

The ID of the base curve that models the shape between two neighboring points.

Compact CCS Power Syntax and Requirements

The expanded, or dynamic, CCS power model syntax provides an important reference and criteria for compact CCS power modeling. See “[Dynamic Current Syntax](#)” on page 17-18 for the `dynamic_current` syntax and see “[Dynamic Power and Ground Current Table Syntax](#)” on page 17-12 for the `pg_current_template` syntax.

The following requirements must be met in the `pg_current_template` group:

- The `last variable_*` value must be `time`. The `time` variable is required.
- The `input_net_transition` and `total_output_net_capacitance` values are available for all `variable_*` attributes except the `last variable_*`. They can be placed in any order except `last`.

The following conditions describe the `pg_current_template` group:

- There can be zero or one `input_net_transition` variable.
- There can be zero, one, or two `total_output_net_capacitance` variables.
- Each `vector` group in the `pg_current` group describes a current waveform that is compacted into a table in the compact CCS power model.

Similar to the compact CCS timing modeling syntax, compact CCS power modeling syntax uses the `base_curves` group to describe normalized base curves and the `compact_lut_template` group as the compact current waveform template. However, the `compact_lut_template` group attributes are extended from three dimensions to four dimensions when CCS power models need two `total_output_net_capacitance` attributes.

The compact CCS power modeling syntax is as follows:

```
library (my_library) {
    base_curves(bc_name) {
        base_curve_type : enum (ccs_half_curve, ccs_timing_half_curve);
        curve_x ("float, ..., float");
        curve_y ("integer, float, ..., float"); /* base curve #1 */
        curve_y ("integer, float, ..., float"); /* base curve #2 */
        ...
        curve_y ("integer, float, ..., float"); /* base curve #n */
    }
}
```

```
compact_lut_template (template_name) {
    base_curves_group : bc_name;
    variable_1 : input_net_transition | total_output_net_capacitance;
    variable_2 : input_net_transition | total_output_net_capacitance;
    variable_3 : input_net_transition | total_output_net_capacitance;
    variable_4 : curve_parameters;
    index_1 ("float, ..., float");
    index_2 ("float, ..., float");
    index_3 ("float, ..., float");
    index_4 ("string, ..., string");
}

...
cell(cell_name) {
    dynamic_current() {
        switching_group() {
            pg_current(pg_pin_name) {
                compact_ccs_power (template_name) {
                    base_curves_group : bc_name;
                    index_output : pin_name;
                    index_1 ("float, ..., float");
                    index_2 ("float, ..., float");
                    index_3 ("float, ..., float");
                    index_4 ("string, ..., string");
                    values ("float | integer, ..., float | integer");
                } /* end of compact_ccs_power */
                ...
            } /* end of pg_current */
            ...
        } /* end of switching_group */
        ...
    } /* end of dynamic_current */
    ...
} /* end of cell */
...
} /* end of library*/
```

Library-Level Groups and Attributes

This section describes library-level groups and attributes used for compact CCS power modeling.

base_curves Group

The `base_curves` group contains the detailed description of normalized base curves. The `base_curves` group has the following attributes:

base_curve_type Complex Attribute

The `base_curve_type` attribute specifies the type of base curve. The `ccs_half_curve` value allows you to model compact CCS power and compact CCS timing data within the same `base_curves` group. You must specify `ccs_half_curve` before specifying `ccs_timing_half_curve`.

curve_x Complex Attribute

The data array contains the x-axis values of the normalized base curve. Only one `curve_x` value is allowed for each `base_curves` group.

For a `ccs_timing_half_curve` base curve, the `curve_x` value must be between 0 and 1 and increase monotonically.

curve_y Complex Attribute

Each base curve consists of one `curve_x` and one `curve_y` attributes. You should define the `curve_x` base curve before `curve_y` for better clarity and easier implementation. The valid region for `curve_y` is [-30, 30] for compact CCS power.

There are two data sections in the `curve_y` complex attribute:

- The `curve_id` integer specifies the identifier of the base curve.
- The data array specifies the y-axis values of the normalized base curve.

compact_lut_template Group

The `compact_lut_template` group is a lookup table template used for compact CCS timing and power modeling.

The following requirements must be met for compact CCS power modeling:

- The last `variable_*` value must be `curve_parameters`.
- The `input_net_transition` and `total_output_net_capacitance` values are available for all `variable_*` attributes except the last `variable_*`. They can be placed in any order except last.

The following conditions describe the `pg_current_template` group:

- There can be zero or one `input_net_transition` variable.
- There can be zero, one, or two `total_output_net_capacitance` variables.
- The element type for all `index_*` values except the last one is a list of floating-point numbers.
- The element type for the last `index_*` value is a string.
- The only valid value for `index_*`, when it is last and when it is specified with `curve_parameters` as the last `variable_*`, is `init_time, init_current, bc_id1, point_time1, point_current1, bc_id2, [point_time2, point_current2, bc_id3, ...], end_time, end_current`.

The valid value in the last index is the pattern that all curve parameter series should follow. It is a pattern rather than a specified series because the table varies in size. Curve parameters define how to describe a current waveform. There should be at least two segments. The reference time for each current waveform is always zero. The negative time values, such as the values with corresponding parameters `init_time`, `point_time` and `end_time`, are permitted.

Note that this index is only for clarity. It is not used to determine the curve parameters. Curve parameters can be uniquely determined by the size of the values. A valid size can be represented as $(8+3i)$, where i is an integer and $i \geq 0$. The current waveform has $(i+2)$ segments.

Cell-Level Groups and Attributes

This section describes cell-level groups and attributes used for compact CCS power modeling.

compact_ccs_power Group

The `compact_ccs_power` group contains a detailed description for compact CCS power data. The `compact_ccs_power` group includes the following optional attributes: `base_curves_group`, `index_1`, `index_2`, `index_3` and `index_4`. The description for these attributes in the `compact_ccs_power` group is the same as in the `compact_lut_template` group. However, the attributes have a higher priority in the `compact_ccs_power` group. For more information, see “[compact_lut_template Group](#)” on page 17-24.

The `index_output` attribute is also optional. It is used only on cross type tables. For more information about the `index_output` attribute, see “Dynamic Current Syntax Checks” in the “Reading and Compiling Libraries” chapter in the *Library Quality Assurance System User Guide*.

values Attribute

The `values` attribute is required in the `compact_ccs_power` group. The data within the quotation marks (" "), or *line*, represent the current waveform for one index combination. Each value is determined by the corresponding curve parameter. In the following line,

```
"t0, c0, 1, t1, c1, 2, t2, c2, 3, t3, c3, 4, t4, c4"
```

the size is $14 = 8+3*2$. Therefore, the curve parameters are as follows:

```
"init_time, init_current, bc_id1, point_time1, point_current1, bc_id2, \
point_time2, point_current2, bc_id3, point_time3, point_current3,
bc_id4,
end_time, end_current"
```

The elements in the `values` attribute are floating-point numbers for time and current and integers for the base curve ID. The number of current waveform segments can be different for each slew and load combination, which means that each line size can be different. As a result, Liberty syntax supports tables with varying sizes, as shown:

```
compact_ccs_power (template_name) {
    ...
    index_1("0.1, 0.2"); /* input_net_transition */
    index_2("1.0, 2.0"); /* total_output_net_capacitance */
    index_3 ("init_time, init_current, bc_id1, point_time1, point_current1,
    bc_id2, [point_time2, point_current2, bc_id3, ...], \
    end_time, end_current"); /* curve_parameters */
    values
    ("t0, c0, 1, t1, c1, 2, t2, c2, 3, t3, c3, 4, t4, c4", /* segment=4 */
```

```

"t0, c0, 1, t1, c1, 2, t2, c2", \           /* segment=2 */
"t0, c0, 1, t1, c1, 2, t2, c2, 3, t3, c3", \ /* segment=3 */
"t0, c0, 1, t1, c1, 2, t2, c2, 3, t3, c3"); /* segment=3 */
}

```

Composite Current Source Dynamic Power Examples

This section provides the following CCS dynamic power examples:

- [Design Cell With a Single Output Example](#)
- [Dense Table With Two Output Pins Example](#)
- [Cross Type With More Than One Output Pin Example](#)
- [Diagonal Type With More Than One Output Pin Example](#)

Design Cell With a Single Output Example

```

pg_current_template ( CCS_power_1 ) {
    variable_1 : input_net_transition ;
    variable_2 : total_output_net_capacitance ;
    variable_3 : time ;
}

cell (example) {
    dynamic_current() {
        when: D;
        related_inputs : CP;
        related_outputs : Q;
        switching_group ( ) {
            input_switching_condition(rise);
            output_switching_condition(rise);
            pg_current (VDD) {
                vector ( CCS_power_1 ) {
                    reference_time : 0.01;
                    index_1 ( 0.01 )
                    index_2 ( 1.0 )
                    index_3 ( 0.000, 0.0873, 0.135, 0.764)
                    values ( 0.002, 0.009, 0.134, 0.546)
                }
            }
        }
    }
}

```

Dense Table With Two Output Pins Example

```

pg_current_template ( CCS_power_1 ) {
    variable_1 : input_net_transition ;
    variable_2 : total_output_net_capacitance ;
    variable_3 : total_output_net_capacitance ;
    variable_4 : time ;
}

cell ( example ) {
    dynamic_current() {
        related_inputs : "A" ;
        related_outputs : "Z" ;
        typical_capacitances(0.04);
        switching_group() {
            input_switching_condition(rise);
            output_switching_condition(rise);
            pg_current(VDD) {
                vector(ccsp_switching_load_time) {
                    reference_time : 0.0015 ;
                    index_1("0.0019");
                    index_2("0.001");
                    index_3("0, 0.006, 0.03, 0.07, 0.09, 0.1, 0.2, 0.3, 0.4,
                           0.5");
                    values("5e-06, 0.001, 0.02, 0.03, 0.05, 0.08, 0.09, 0.04,
                           0.009, 5.0e-06");
                }
            }
        }
    }
}

```

Cross Type With More Than One Output Pin Example

```

pg_current_template ( CCS_power_1 ) {
    variable_1 : input_net_transition ;
    variable_2 : total_output_net_capacitance ;
    variable_3 : time ;
}

cell ( example )

dynamic_current() {
    when: D;
        related_inputs : CP;
        related_outputs : Q QN QN1 QN2;
        typical_capacitances(10.0 10.0 10.0 10.0);
        switching_group () {
            input_switching_condition(rise);
            output_switching_condition(rise, fall, fall, fall);

```

```
pg_current (VSS) {
    vector ( CCS_power_1 ) {
        index_output : Q;
        reference_time : 0.01;
        index_1 ( 0.01 )
        index_2 ( 5.0 )
        index_3 ( 0.000, 0.0873, 0.135, 0.764)
        values ( 0.002, 0.009, 0.134, 0.546)
    }

    vector ( CCS_power_1 ) {
        index_output : QN;
        reference_time : 0.01;
        index_1 ( 0.01 )
        index_2 ( 1.0 )
        index_3 ( 0.000, 0.0873, 0.135, 0.764)
        values ( 0.002, 0.009, 0.134, 0.546)
    }

    vector ( CCS_power_1 ) {
        index_output : QN;
        reference_time : 0.01;
        index_1 ( 0.01 )
        index_2 ( 5.0 )
        index_3 ( 0.000, 0.0873, 0.135, 0.764)
        values ( 0.002, 0.009, 0.134, 0.546)
    }
}
```

Diagonal Type With More Than One Output Pin Example

```
pg_current_template ( CCS_power_1 ) {
    variable_1 : input_net_transition ;
    variable_2 : total_output_net_capacitance ;
    variable_3 : time ;
}
cell ( example )
    dynamic_current() {
        when: D ;
        related_inputs : CP;
        related_outputs : Q QN QN1 QN2;
        switching_group ( ) {
            input_switching_condition(rise);
            output_switching_condition(rise, fall, fall, fall);
        }
    }
    pg_current (VSS) {
        vector ( CCS_power_1 ) {
            reference_time : 0.01;
            index_1 ( 0.01 )
            index_2 ( 1.0 )
            index_3 ( 0.000, 0.0873, 0.135, 0.764)
            values ( 0.002, 0.009, 0.134, 0.546)
        }
    }
}
```


18

On-Chip Variation (OCV) Modeling

On-chip variation (OCV) causes variation in the timing performance of each transistor in a design. The Library Compiler tool supports advanced and parametric OCV modeling.

In advanced OCV models, these variations are modeled using derating factors based on the path depth and path distance.

In parametric OCV models, the variations are modeled using random variation and path distance of cells. The Liberty variation format (LVF) is used to represent the random variation library data. The PrimeTime tool can use the models during OCV analysis.

The OCV model definitions are covered in the following sections:

- [OCV Model Overview](#)
- [Advanced OCV Modeling](#)
- [Parametric OCV Liberty Variation Format Modeling](#)

OCV Model Overview

[Table 18-1](#) shows an overview of the OCV models supported by the Library Compiler tool. Path depth is the position of a cell in a delay path, and path distance is the length of the path.

Table 18-1 Library OCV Modeling Overview

		Advanced OCV	Parametric OCV
		Distance based table	Sigma based table
Content	Table of derating factors based on path depth, or distance, or both	Table of derating factors based on path distance	Table of random variation values based on statistical deviation
Scope	Library, cell, or both	Library, cell, or both	Cell, timing arc, or both
Values	Derating factor	Derating factor	Absolute variation offset
Support for timing constraints	No	No	Yes
Scalar value	Yes	Yes	Yes

Advanced OCV Modeling

In an advanced OCV model, the derating values are based on the path depth and path distance.

Note:

The derating values are used during advanced OCV analysis, which is less pessimistic than a conventional OCV analysis.

For more information about advanced OCV analysis, see the *PrimeTime User Guide*.

The modeling syntax is:

```
library (name) {
  ocv_table_template(ocv_template_name) {
```

```

variable_1: path_depth|path_distance;
variable_2: path_depth|path_distance;
index_1 ("float..., float");
index_2 ("float..., float");
}
ocv_derate(ocv_derate_group_name) {
    ocv_derate_factors(ocv_template_name) {
        rf_type: rise|fall|rise_and_fall;
        derate_type: early|late;
        path_type: clock|data|clock_and_data;
        index_1 ("float,..., float");
        index_2 ("float,..., float");
        values ( "float,..., float", \
                  ..., \
                  "float,..., float");
    }
    ...
}
default_ocv_derate_group: ocv_derate_group_name;
ocv_arc_depth: float;
...
cell (cell_name) {
    ocv_derate_group: ocv_derate_group_name;
    ocv_arc_depth: float;
    ...
    ocv_derate(ocv_derate_group_name) {
        ocv_derate_factors(ocv_template_name) {
            rf_type: rise|fall|rise_and_fall;
            derate_type: early|late;
            path_type: clock|data|clock_and_data;
            index_1 ("float,..., float");
            index_2 ("float,..., float");
            values ( "float,..., float", \
                      ..., \
                      "float,..., float");
        }
        ...
    }
    ...
}
pin | bus | bundle (name) {
    direction: inout | output;
    timing() {
        ocv_arc_depth: float;
        ...
    } /* end of timing */
} /* end of inout | output pin */
...
} /* end of cell */
...
} /* end of library */

```

Table 18-2 shows the association between the Liberty advanced OCV syntax and the PrimeTime file format for advanced OCV analysis.

Table 18-2 Association Between the Liberty Advanced OCV Syntax and the PrimeTime Advanced OCV File Format

Liberty syntax	PrimeTime advanced OCV file format
ocv_derate	group_name
ocv_table_template	table
path_depth	depth
path_distance	distance
ocv_derate_factors	table
rf_type	rf_type
derate_type	derate_type
path_type	path_type
ocv_derate_group	group_name
default_ocv_derate_group	group_name
ocv_arc_depth	Not applicable

Library-Level Groups and Attributes

This section describes library-level groups and attributes for advanced OCV modeling that also apply to cells and timing arcs where specified.

ocv_table_template Group

The `ocv_table_template` group defines the template for an `ocv_derate_factors` group defined within the `ocv_derate` group.

The template definition includes two one-dimensional variables, `variable_1` and `variable_2`. These variables are used as indexes in the `ocv_derate_factors` group. You must specify `variable_1`. Specifying `variable_2` is optional.

Both `variable_1` and `variable_2` can have one of the following values:

- `path_depth`: The number of cells in a delay path.
- `path_distance`: The physical distance spanned by the path. Use the `distance_unit` attribute in the library to specify the path distance unit.

You can specify the following types of lookup tables within the `ocv_table_template` group:

- One-dimensional tables with the index consisting of `path_depth` or `path_distance` for a table
- Two-dimensional tables with one index consisting of `path_depth`, and the other index consisting of `path_distance`

This means that for a two-dimensional table, the values of `variable_1` and `variable_2` are different.

`ocv_derate` Group

The `ocv_derate` group contains a set of `ocv_derate_factors` groups. The `ocv_derate` group specifies a set of lookup tables with OCV derating factors for timing.

You must specify at least one `ocv_derate_factors` group within an `ocv_derate` group.

You can specify the `ocv_derate` group in the `library` and `cell` groups. To define multiple `ocv_derate` groups in a library, use different group names. If multiple `ocv_derate` groups have the same name, the group that is last specified overrides the previous ones.

`ocv_derate_factors` Group

The `ocv_derate_factors` group specifies a lookup table of the OCV derating factors. The lookup table can be one-dimensional, two-dimensional, or scalar. For a one-dimensional lookup table, the index consists of `path_depth` or `path_distance` values. For a two-dimensional lookup table, the indexes consist of the `path_depth` and `path_distance` values. Use the scalar to specify a single derating factor irrespective of the path depth and path distance.

Use the following attributes to define specific OCV derating factors in the `ocv_derate_factors` lookup table:

- `rf_type`

The `rf_type` attribute defines the type of delay specified in the `ocv_derate_factors` lookup table. Valid values are `rise`, `fall`, and `rise_and_fall`.

- `derate_type`

The `derate_type` attribute defines the type of arrival time specified in the `ocv_derate_factors` lookup table. The valid values are `early` and `late`.

- `path_type`

The `path_type` attribute defines the type of path specified in the `ocv_derate_factors` group. The valid values are `clock`, `data`, and `clock_and_data`.

These attributes do not have defaults. You must specify these attributes in the `ocv_derate_factors` group.

Applying OCV Derating Factors to Library Cells

To apply the set of derating factors defined within an `ocv_derate` group to a cell, use the cell-level `ocv_derate_group` attribute to specify the name of the `ocv_derate` group.

To apply the same set of derating factors to multiple cells in a library, specify the `ocv_derate` group at the library-level and define the name of the `ocv_derate` group with the `ocv_derate_group` attribute in the corresponding `cell` groups.

The set of derating factors defined within a cell-level `ocv_derate` group can be applied only to the particular cell.

If you do not specify the `ocv_derate_group` attribute, the default `ocv_derate` group defined at the library level applies to the cell.

default_ocv_derate_group Attribute

The `default_ocv_derate_group` attribute specifies the name of the default `ocv_derate` group for a library. The default `ocv_derate` group applies to the `cell` groups where the `ocv_derate` group name is not defined using the cell-level `ocv_derate_group` attribute.

ocv_arc_depth Attribute

The optional `ocv_arc_depth` attribute specifies the effective logic depth of a cell or a timing arc. The default is 1.0.

Tools, such as PrimeTime can use the value of this attribute to calculate the total effective logic depth of the path that includes the cell or timing arc. The path depth is used to determine the OCV derating factors from the lookup tables in the `ocv_derate_factors` group.

You can define the `ocv_arc_depth` attribute in the `library`, `cell`, or `timing` groups. If you define this attribute in different groups, the attribute value in the `timing` group overrides the value defined in the `cell` group, and the value defined in the `cell` group overrides the value defined in the `library` group.

Cell-Level Attribute

This section describes a cell-level attribute for advanced OCV modeling.

ocv_derate_group Attribute

The `ocv_derate_group` attribute specifies the name of the `ocv_derate` group that applies to a cell.

Advanced OCV Library Example

[Example 18-1](#) shows a library modeled for advanced OCV.

Example 18-1 A Typical Advanced OCV Library

```
library (OCV_lib) {
    ocv_table_template(2D_ocv_template) {
        variable_1: path_depth;
        variable_2: path_distance;
        index_1 ("1,2,3");
        index_2 ("1,2,3");
    }
    ocv_table_template(1D_ocv_template1) {
        variable_1: path_depth;
        index_1 ("1,2,3");
    }
    ocv_table_template(1D_ocv_template2) {
        variable_1: path_distance;
        index_1 ("1,2,3");
    }
    ocv_derate(advanced_ocv){
        ocv_derate_factors(scalar) {
            rf_type: rise_and_fall;
            derate_type: early;
            path_type: clock_and_data;
            values ( "0.100");
        }
        ocv_derate_factors(scalar) {
            rf_type: rise_and_fall;
            derate_type: late;
            path_type: clock_and_data;
            values ( "0.200");
        }
    }
    ocv_arc_depth: 1.0;
    default_ocv_derate_group: advanced_ocv;
    cell (cell1) {
        ocv_arc_depth: 2.0;
        ocv_derate_group: aocv1;
```

```

ocv_derate(aocv1) {
    ocv_derate_factors (2D_ocv_template) {
        rf_type: rise_and_fall;
        derate_type: early;
        path_type: clock_and_data;
        index_1 ("1,2,3");
        index_2 ("4, 5");
        values ( "0.1, 0.2, 0.3",
                  "0.2, 0.3, 0.4");
    }
    ocv_derate_factors (2D_ocv_template) {
        rf_type: rise_and_fall;
        derate_type: late;
        path_type: clock_and_data;
        index_1 ("1,2,3");
        index_2 ("100, 200");
        values ( "0.1, 0.2, 0.3",
                  "0.2, 0.3, 0.4");
    }
}
ocv_derate(aocv2) {
    ocv_derate_factors(scalar) {
        rf_type: rise_and_fall;
        derate_type: late;
        path_type: clock_and_data;
        values ( "0.900");
    }
    ocv_derate_factors(scalar) {
        rf_type: rise_and_fall;
        derate_type: early;
        path_type: clock_and_data;
        values ( "0.800");
    }
}
...
pin | bus | bundle (name) {
    direction: inout | output;
    timing() {
        ocv_arc_depth: 3.0;
        ...
    } /* end of timing */
}
...
} /* end of cell */
cell (cell12) {
    ocv_derate_group: advanced_ocv;
    ...
} /* end of cell */
cell (cell13) {
    /* use advanced_ocv as default_ocv_derate_group
       defined in library */
    ...
} /* end of cell */

```

```
    } ... /* end of library */
```

Advanced OCV Checks

The Library Compiler tool checks the advanced OCV syntax and issues errors or warning messages if certain conditions occur. For a detailed list of these library checks, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter of the *Library Quality Assurance System User Guide*.

Parametric OCV Liberty Variation Format Modeling

In a parametric OCV (POCV) model, the random variation is specific to a cell or a timing arc, unlike an advanced OCV model where a specific derating factor applies to multiple cells or an entire library. The Liberty variation format (LVF) is used to represent the parametric OCV (POCV) library data, such as the slew-load table per delay timing arc. The Liberty variation format (LVF) syntax consists of groups for sigma variation cell delay, output transition or slew, and constraint tables that are load and input-slew dependent. The variation values are used during parametric OCV analysis.

The parametric OCV Liberty variation format (LVF) modeling syntax is:

```
library (name) {
    lu_table_template(lu_template_name) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        variable_3: related_out_total_output_net_capacitance;
        index_1 ("float, ..., float");
        index_2 ("float, ..., float");
        index_3 ("float, ..., float");
    }

    lu_table_template(constraint_lu_template_name) {
        variable_1: related_pin_transition;
        variable_2: constrained_pin_transition;
        variable_3: related_out_total_output_net_capacitance | \
                    related_out_output_net_length | \
                    related_out_output_net_wire_cap | \
                    related_out_output_net_pin_cap;
        index_1 ("float, ..., float");
        index_2 ("float, ..., float");
        index_3 ("float, ..., float");
    }

    ocv_table_template(ocv_template_name) {
        variable_1: path_distance;
        index_1 ("float, ..., float");
    }
}
```

```

ocv_derate(ocv_derate_group_name) {
    ocv_derate_factors(ocv_template_name) {
        rf_type: rise | fall | rise_and_fall;
        derate_type: early | late;
        path_type: clock | data | clock_and_data;
        index_1 ("float, ..., float");
        values ( "float, ..., float");
    }
    ...
}
default_ocv_derate_distance_group: ocv_derate_group_name;
...
cell (cell_name) {
    ocv_derate_distance_group: ocv_derate_group_name;
    ...
    pin | bus | bundle (name) {
        direction: input | output;
        timing() {
            ...
            ocv_sigma_cell_rise(lu_template_name){
                sigma_type: early | late | early_and_late;
                index_1 ("float, ..., float");
                index_2 ("float, ..., float");
                values ( "float, ..., float", \
                    ...,\n
                    "float, ..., float");
            }
            ocv_sigma_cell_fall(lu_template_name){
                sigma_type: early | late | early_and_late;
                index_1 ("float, ..., float");
                index_2 ("float, ..., float");
                values ( "float, ..., float", \
                    ...,\n
                    "float, ..., float");
            }
            ocv_sigma_rise_transition(lu_template_name){
                sigma_type: early | late | early_and_late;
                index_1 ("float, ..., float");
                index_2 ("float, ..., float");
                index_3 ("float, ..., float");
                values ( "float, ..., float", \
                    ...,\n
                    "float, ..., float");
            }
            ocv_sigma_fall_transition(lu_template_name){
                sigma_type: early | late | early_and_late;
                index_1 ("float, ..., float");
                index_2 ("float, ..., float");
                index_3 ("float, ..., float");
                values ( "float, ..., float", \
                    ...,\n
                    "float, ..., float");
            }
        }
    }
}

```

```

    ...
} /* end of timing */
...
} /* end of pin */
...
pin | bus | bundle (name) {
  direction: input | inout;
  timing() {
    ...
    ocv_sigma_rise_constraint(constraint_lu_template_name) {
      index_1 ("float, ..., float");
      index_2 ("float, ..., float");
      values ( "float, ..., float", \
                ...,\ \
                  "float, ..., float");
    }
    ocv_sigma_fall_constraint(constraint_lu_template_name) {
      index_1 ("float, ..., float");
      index_2 ("float, ..., float");
      values ( "float, ..., float", \
                ...,\ \
                  "float, ..., float");
    }
    ...
} /* end of timing */
...
} /* end of pin */
...
} /* end of cell */
...
} /* end of library */

```

Note:

Do not specify the `sigma_type` attribute in the `ocv_sigma_rise_constraint` and `ocv_sigma_fall_constraint` groups. Otherwise, the tool generates an error message.

Table 18-3 shows the association between the Liberty parametric OCV syntax and the PrimeTime file format for parametric OCV analysis.

Table 18-3 Association Between the Liberty Parametric OCV Syntax and the PrimeTime Parametric OCV File Format

Liberty syntax	PrimeTime parametric OCV file format
<code>ocv_derate</code>	<code>group_name</code>
<code>ocv_table_template</code>	<code>table</code>
<code>path_distance</code>	<code>distance</code>
<code>ocv_derate_factors</code>	<code>table</code>

Table 18-3 Association Between the Liberty Parametric OCV Syntax and the PrimeTime Parametric OCV File Format (Continued)

Liberty syntax	PrimeTime parametric OCV file
rf_type	rf_type
derate_type	derate_type
path_type	path_type
ocv_derate_distance_group	group_name
default_ocv_derate_distance_group	group_name

Library-Level Groups and Attributes

This section describes library-level groups and attributes for parametric OCV modeling that also apply to cells and timing arcs where specified.

lu_table_template Group

The `lu_table_template` group defines the template for the following groups specified in the `timing` group:

- `ocv_sigma_cell_rise`
- `ocv_sigma_cell_fall`
- `ocv_sigma_rise_transition`
- `ocv_sigma_fall_transition`
- `ocv_sigma_rise_constraint`
- `ocv_sigma_fall_constraint`

For the `ocv_sigma_cell_rise`, `ocv_sigma_cell_fall`, `ocv_sigma_rise_transition`, and `ocv_sigma_fall_transition` groups, the template definition includes the `variable_1`, `variable_2`, `variable_3`, `index_1`, `index_2`, and `index_3` attributes. The `variable_1`, `variable_2` and `variable_3` attributes specify the index variables for the lookup tables (LUTs) in these groups. The `index_1`, `index_2`, and `index_3` attributes specify the numerical values of the variables.

The values of `variable_1`, `variable_2`, and `variable_3` attributes are `input_net_transition`, `total_output_net_capacitance`, and `related_out_total_output_net_capacitance`.

You can specify the following types of lookup tables within these groups:

- One-dimensional tables with a single index, such as `input_net_transition`
- Two-dimensional tables with the indexes, `input_net_transition` and `total_output_net_capacitance`
- Three-dimensional tables with the indexes, `input_net_transition`, `total_output_net_capacitance`, and `related_out_total_output_net_capacitance`
- A scalar that has a single variation value irrespective of the `input_net_transition`, `total_output_net_capacitance`, and `related_out_total_output_net_capacitance` values

For the `ocv_sigma_rise_constraint` and `ocv_sigma_fall_constraint` groups, the template definition includes the `variable_1`, `variable_2`, `variable_3`, `index_1`, `index_2`, and `index_3` attributes. The `variable_1`, `variable_2` and `variable_3` attributes specify the index variables for the lookup tables of the `ocv_sigma_rise_constraint` and `ocv_sigma_fall_constraint` groups. The `index_1`, `index_2`, and `index_3` attributes specify the numerical values of these variables.

The values of `variable_1` and `variable_2` are `related_pin_transition` and `constrained_pin_transition`, respectively. The values of `variable_3` can be `related_out_total_output_net_capacitance`, `related_out_output_net_length`, `related_out_net_wire_cap`, or `related_out_output_net_pin_cap`. You can specify the following types of lookup tables within these groups:

- One-dimensional tables with a single index, such as `related_pin_transition`
- Two-dimensional tables with the indexes, `related_pin_transition` and `constrained_pin_transition`
- Three-dimensional tables with the indexes, `related_pin_transition`, `constrained_pin_transition`, and a valid value of `variable_3`
- A scalar that has a single variation value irrespective of the `related_pin_transition`, `constrained_pin_transition`, and `variable_3` values

ocv_table_template Group

The `ocv_table_template` group defines the template for an `ocv_derate_factors` group defined within the `ocv_derate` group.

The template definition includes a one-dimensional variable, `variable_1`, that is used as an index in the `ocv_derate_factors` group. Valid value of `variable_1` is `path_distance` and represents the physical distance spanned by the path. Use the `distance_unit` attribute in the library to specify the path distance unit.

ocv_derate Group

The `ocv_derate` group contains a set of `ocv_derate_factors` groups. The `ocv_derate` group specifies a set of lookup tables with OCV derating factors for timing.

You must specify at least one `ocv_derate_factors` group within an `ocv_derate` group.

You can specify the `ocv_derate` group in the `library` and `cell` groups. To define multiple `ocv_derate` groups in a library, use different group names. If multiple `ocv_derate` groups have the same name, the group that is last-specified overrides the previous ones.

ocv_derate_factors Group

The `ocv_derate_factors` group specifies a lookup table of the OCV derating factors. The lookup table can be one-dimensional or scalar. For a one-dimensional lookup table, the index consists of `path_distance` values. Use the scalar to specify a single derating factor irrespective of the path distance.

Use the following attributes to define specific OCV derating factors in the `ocv_derate_factors` lookup table:

- `rf_type`

The `rf_type` attribute defines the type of delay specified in the `ocv_derate_factors` lookup table. Valid values are `rise`, `fall`, and `rise_and_fall`.

- `derate_type`

The `derate_type` attribute defines the type of arrival time specified in the `ocv_derate_factors` lookup table. The valid values are `early` and `late`.

- `path_type`

The `path_type` attribute defines the type of path specified in the `ocv_derate_factors` group. The valid values are `clock`, `data`, and `clock_and_data`.

These attributes do not have defaults. You must specify these attributes in the `ocv_derate_factors` group.

Applying Parametric OCV Derating Factors to Library Cells

To apply the set of derating factors defined within an `ocv_derate` group to a cell, use the cell-level `ocv_derate_distance_group` attribute to specify the name of the `ocv_derate` group.

To apply the same set of derating factors to multiple cells in a library, specify the `ocv_derate` group at the library-level and define the name of the `ocv_derate` group with the `ocv_derate_distance_group` attribute in the corresponding `cell` groups.

The set of derating factors defined within a cell-level `ocv_derate` group can be applied only to the particular cell.

If you do not specify the `ocv_derate_distance_group` attribute, the default `ocv_derate` group defined at the library level applies to the cell.

default_ocv_derate_distance_group Attribute

The `default_ocv_derate_group` attribute specifies the name of the default `ocv_derate` group for a library. The default `ocv_derate` group applies to the cell groups where the `ocv_derate` group name is not defined using the cell-level `ocv_derate_distance_group` attribute.

Cell-Level Attribute

This section describes a cell-level attribute for parametric OCV modeling.

ocv_derate_distance_group Attribute

The `ocv_derate_distance_group` attribute specifies the name of the `ocv_derate` group that applies to a cell.

Arc-Level Groups and Attributes

This section describes the timing arc-level groups for modeling delay variations. Each of the groups specify a lookup table. The lookup table can be one-dimensional, two-dimensional, three-dimensional, or scalar.

Define the template of the lookup table in the library-level `lu_table_template` group. To use the template, specify the name of the `lu_table_template` group as the arc-level group name.

ocv_sigma_cell_rise Group

The `ocv_sigma_cell_rise` group specifies a lookup table for the rise delay variation. In the lookup table, each absolute rise-delay variation offset from the corresponding nominal rise delay is specified at one sigma (σ), where sigma is the standard deviation of the rise delay distribution.

Note:

The nominal rise delay value is specified by the `cell_rise` group within the `timing` group. For more information, see [Chapter 11, “Timing Arcs”](#).

During parametric OCV analysis, the delay variation and the nominal rise delay are used to calculate the rise delay at one sigma (σ):

```
rise delay(±s) = nominal rise delay ± ocv_sigma_cell_rise value
```

ocv_sigma_cell_fall Group

The `ocv_sigma_cell_fall` group specifies a lookup table for the fall delay variation. In the lookup table, each absolute fall-delay variation offset from the corresponding nominal fall delay is specified at one sigma (σ), where sigma is the standard deviation of the fall delay distribution.

Note:

The nominal fall delay value is specified by the `cell_fall` group within the `timing` group. For more information, [Chapter 11, “Timing Arcs”](#).

During OCV analysis, the delay variation and the nominal fall delay are used to calculate the fall delay at one sigma (σ):

```
fall delay(±s) = nominal fall delay ± ocv_sigma_cell_fall value
```

ocv_sigma_rise_transition Group

The `ocv_sigma_rise_transition` group specifies a lookup table of the rise transition variation values. In the lookup table, each absolute rise-transition variation offset from the corresponding nominal rise transition is specified at one sigma (σ), where sigma is the standard deviation of the rise transition distribution.

Note:

The nominal rise transition value is specified by the `rise_transition` group within the `timing` group. For more information, see [Chapter 11, “Timing Arcs”](#).

During parametric OCV analysis, the rise transition variation and the nominal rise transition are used to calculate the rise transition at one sigma (σ):

```
rise transition(±s) = nominal rise transition ± ocv_sigma_rise_transition value
```

ocv_sigma_fall_transition Group

The `ocv_sigma_fall_transition` group specifies a lookup table for the fall transition variation. In the lookup table, each absolute fall-transition variation offset from the nominal fall transition is specified at one sigma (σ), where sigma is the standard deviation of the fall transition distribution.

Note:

The nominal fall transition value is specified by the `fall_transition` group within the `timing` group. For more information, [Chapter 11, “Timing Arcs”](#).

During parametric OCV analysis, the transition variation and the nominal fall transition are used to calculate the fall delay at one sigma (σ):

```
fall transition(±s) = nominal fall transition ± ocv_sigma_fall_transition
value
```

sigma_type Attribute

The optional `sigma_type` attribute defines the type of arrival time specified in the `ocv_sigma_cell_rise`, `ocv_sigma_cell_fall`, `ocv_sigma_rise_transition`, and `ocv_sigma_fall_transition` lookup tables. The values are `early`, `late`, and `early_and_late`. The default is `early_and_late`.

During parametric OCV analysis, when you specify the `sigma_type` attribute in these groups, the following values at $\pm\sigma$ are calculated as:

```
rise delay(+s) = nominal cell rise + ocv_sigma_cell_riselate
rise delay(-s) = nominal cell rise - ocv_sigma_cell_riseearly
fall delay(+s) = nominal cell fall + ocv_sigma_cell_falllate
fall delay(-s) = nominal cell fall - ocv_sigma_cell_fallearly
```

```
rise transition(+s) = nominal rise transition + ocv_sigma_rise_transitionlate
rise transition(-s) = nominal rise transition - ocv_sigma_rise_transitionearly
fall transition(+s) = nominal fall transition + ocv_sigma_fall_transitionlate
fall transition(-s) = nominal fall transition - ocv_sigma_fall_transitionearly
```

ocv_sigma_rise_constraint Group

The `ocv_sigma_rise_constraint` group specifies a lookup table of the rise constraint variation values. In the lookup table, each absolute rise-constraint variation offset from the nominal rise constraint is specified at one sigma (σ), where sigma is the standard deviation of the rise constraint distribution.

Note:

The nominal rise constraint value is specified by the `rise_constraint` group within the `timing` group. For more information, see [Chapter 11, “Timing Arcs”](#).

During OCV analysis, the rise constraint variation and the nominal rise constraint are used to calculate the rise constraint at one sigma (σ):

```
rise constraint(±s) = nominal rise constraint ± ocv_sigma_rise_constraint
value
```

ocv_sigma_fall_constraint Group

The `ocv_sigma_fall_constraint` group specifies a lookup table for the fall constraint variation. In the lookup table, each absolute fall-constraint variation offset from the nominal fall constraint is specified at one sigma (σ), where sigma is the standard deviation of the fall constraint distribution.

Note:

The nominal fall constraint value is specified by the `fall_constraint` group within the `timing` group. For more information, [Chapter 11, “Timing Arcs”](#).

During parametric OCV (POCV) analysis, the delay variation and the nominal fall delay are used to calculate the fall delay at one sigma (σ):

```
fall_constraint(±s) = nominal fall constraint ± ocv_sigma_fall_constraint
value
```

Parametric OCV Library Example

[Example 18-2](#) shows a library description of the variation model.

Example 18-2 A Library Model for OCV Delays, Transitions, and Constraints

```
library (lib1) {
    lu_table_template(2D_lu_template) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        index_1 ("1,2,3");
        index_2 ("1,2,3");
    }
    lu_table_template(3D_lu_template) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        variable_3: related_out_total_output_net_capacitance;
        index_1 ("1,2,3");
        index_2 ("1,2,3");
        index_3 ("1,2,3");
    }
    lu_table_template(2D_constraint_lu_template) {
        variable_1: related_pin_transition;
        variable_2: constrained_pin_transition;
        index_1 ("1,2,3");
        index_2 ("1,2,3");
    }
    ocv_table_template(1D_ocv_template2) {
        variable_1: path_distance;
        index_1 ("1,2,3");
    }
    ocv_derate(location_based_ocv) {
        ocv_derate_factors(1D_ocv_template2) {
            rf_type: rise_and_fall;
            derate_type: early;
            path_type: clock_and_data;
            index_1 ("1, 2");
            values ( "5.0, 6.0");
        }
    }
    ocv_derate(locv1) {
```

```

ocv_derate_factors(1D_ocv_template2) {
    rf_type: rise_and_fall;
    derate_type: early;
    path_type: clock_and_data;
    index_1 ("1, 2");
    values ( "5.0, 6.0");
}
}
default_ocv_derate_distance_group: location_based_ocv;
cell (cell1) {
    ocv_derate_distance_group: locv1;
    ...
    pin (pin1) {
        direction: output;
        timing() {
            ...
            related_pin : "rpin1" ;
            timing_type : rising_edge ;
            cell_rise( 2D_lu_template ){
                index_1 ( "0.1, 0.2, 0.3");
                index_2 ( "0.4, 0.5, 0.6");
                values ( "0.1, 0.2, 0.3",
                         "0.2, 0.3, 0.4",
                         "0.3, 0.4, 0.5");
            }
            cell_fall( scalar ){
                values ("0.100");
            }
            rise_transition( 2D_lu_template ){
                index_1 ( "0.1, 0.2, 0.3");
                index_2 ( "0.4, 0.4, 0.5");
                values ( "0.1, 0.2, 0.3",
                         "0.3, 0.4, 0.5",
                         "0.5, 0.6, 0.7");
            }
            fall_transition( scalar ){
                values ("0.100");
            }
            ocv_sigma_cell_rise( 2D_lu_template ){
                sigma_type: early;
                index_1 ( "0.3, 0.4, 0.5");
                index_2 ( "0.1, 0.2, 0.3");
                values ( "0.1, 0.2, 0.3",
                         "0.2, 0.3, 0.4",
                         "0.3, 0.4, 0.5");
            }
            ocv_sigma_cell_rise( 2D_lu_template ){
                sigma_type: late;
                index_1 ( "0.3, 0.4, 0.5");
                index_2 ( "0.1, 0.2, 0.3");
                values ( "0.1, 0.2, 0.3",
                         "0.2, 0.3, 0.4",
                         "0.3, 0.4, 0.5");
            }
        }
    }
}

```

```

        }
        ocv_sigma_cell_fall( scalar ){
            sigma_type: early;
            values ("0.1");
        }
        ocv_sigma_cell_fall( scalar ){
            sigma_type: late;
            values ("0.1");
        }
        ocv_sigma_rise_transition( 3D_lu_template ){
            sigma_type: early;
            index_1 ( "0.1, 0.2, 0.3");
            index_2 ( "0.4, 0.5, 0.6");
            index_3 ( "0.7, 0.8");
            values ("0.1, 0.2, 0.3",\
                    "0.2, 0.3, 0.4",\
                    "0.3, 0.4, 0.5",\
                    "0.4, 0.5, 0.6",\
                    "0.5, 0.6, 0.7",\
                    "0.6, 0.7, 0.8");
        }
        ocv_sigma_rise_transition( 3D_lu_template ){
            sigma_type: late;
            index_1 ( "0.1, 0.2, 0.3");
            index_2 ( "0.4, 0.5, 0.6");
            index_3 ( "0.7, 0.8");
            values ("0.1, 0.2, 0.3",\
                    "0.2, 0.3, 0.4",\
                    "0.3, 0.4, 0.5",\
                    "0.4, 0.5, 0.6",\
                    "0.5, 0.6, 0.7",\
                    "0.6, 0.7, 0.8");
        }
        ocv_sigma_fall_transition( scalar ){
            sigma_type: early_and_late;
            values ( "0.200");
        }
        ...
    } /* end of timing */
}
...
pin (pin2) {
    direction: input | inout;
    timing() {
        ...
        ocv_sigma_rise_constraint( 2D_lu_constraint_template ){
            index_1( "0.3, 0.4, 0.5");
            index_2 ( "0.1, 0.2, 0.3");
            values ( "0.1, 0.2, 0.3",\
                    "0.2, 0.3, 0.4",\
                    "0.3, 0.4, 0.5");
        }
        ocv_sigma_fall_constraint( 2D_lu_constraint_template ){

```

```
    index_1 ( "0.3, 0.4, 0.5");
    index_2 ( "0.1, 0.2, 0.3");
    values ( "0.1, 0.2, 0.3", \
              "0.2, 0.3, 0.4", \
              "0.3, 0.4, 0.5");
}
...
} /* end of timing */
...
} /* end of pin */
...
} /* end of cell */
cell (cell2) {
/* use location_based_ocv as
   default_ocv_derate_distance_group defined in library */
...
} /* end of cell */
...
} /* end of library */
```

Parametric OCV Checks

The Library Compiler tool checks the parametric OCV syntax and issues errors or warning messages if certain conditions occur. For a detailed list of these library checks, see “Library Screener Checks” in the “Reading and Compiling Libraries” chapter of the *Library Quality Assurance System User Guide*.

19

Defining Library Model Files

A Liberty library includes various types of data. Each of the data types can have specific models, such as nonlinear delay models (NLDMs) and composite current source (CCS) timing models, nonlinear power models (NLPMs) and CCS power models, and Liberty variation format (LVF) models.

A base library file is a .lib file that includes the standard Liberty syntax necessary to model all the relevant data types and different models for each data type. The base library is compiled and written out to generate a library database, which is a .db file.

A Liberty model file is an incremental .lib file compiled for use with the corresponding base library during special design flows. The Liberty model file is compiled and written out to generate a model file database, which is also a .db file that contains the specific models defined in the source Liberty model file.

This chapter describes the following concepts and tasks:

- [Model Files](#)
- [Model File Syntax](#)
- [Compiling Model Files](#)

Model Files

A Liberty model file contains the following syntax categories:

- Model syntax
 - The Liberty syntax for one or more specific models
- Reference syntax
 - Information required to compile, such as, the type of bus
 - Information required to pass the model syntax screener checks
- Framework syntax
 - Parent groups that contain the model syntax, reference syntax, and syntax that links the model file to the base library, such as, the `library`, `cell`, and `pin` groups

The values of the attributes in the base library and the model file can be different. If the model syntax values are different in the model file, this data is used to update the data in the base library. If the reference syntax values are different in the model file, this is an error and you must correct it.

Types of Model Files

The following model file types are supported:

- Liberty variation format (LVF) that includes the following types:
 - Advanced OCV
 - Parametric OCV
- CCS power
- Power aware

Power-Aware Model File

A power-aware model file contains IEEE 1801, also known as the United Power Format (UPF), related Liberty syntax. Use the power-aware model file to update a base library for power analysis. The model file is corner-independent.

Table 19-1 Examples of Syntax Categories in a Power-Aware Model File

Model file syntax categories	Examples
Model	PG pin and power management information, such as, the <code>input_voltage_range</code> attribute for a level-shifter cell
Reference	Information required to compile, such as, the type of bus and referenced signal pin groups for bus bit-blasting Information required to pass the model syntax screener checks, such as, the <code>pin direction</code> attribute to check if the <code>input_voltage_range</code> is defined in an input pin and not in an output pin
Framework	Parent groups that contain the model syntax, reference syntax, and syntax that links the model file to the base library, such as, the <code>library</code> , <code>cell</code> , and <code>pin</code> groups

The following table shows the syntax content of a power-aware model file for different types of base libraries, such that together they contain the complete power-aware syntax.

Base library content	Model file content
PG pin definition	Complete or additional power management attributes
Partial and correct power management attributes	
PG pin definition	Complete or additional and correct power management attributes
Partial and some incorrect power management attributes	
PG pin definition	Complete list of <code>pg_pin</code> groups with defined <code>pg_type</code> attributes
No power management attributes	Complete list of <code>pin</code> groups with defined <code>direction</code> attributes Complete set of power management attributes

Model File Syntax

The following sections list the syntax for the supported model types:

- [Liberty Variation Format Advanced OCV Model Syntax](#)
 - [Liberty Variation Format Parametric OCV Model Syntax](#)
 - [CCS Power Model Syntax](#)
 - [Power-Aware Model Syntax](#)
 - [Reference Syntax](#)
 - [Framework Syntax For Power-Aware Model Files](#)
-

Liberty Variation Format Advanced OCV Model Syntax

```
library (library_name) {
    ocv_table_template(ocv_template_name) {
        variable_1: path_depth | path_distance;
        variable_2: path_depth | path_distance;
        index_1 ("float..., float");
        index_2 ("float..., float");
    }
    ocv_derate(ocv_derate_group_name) {
        ocv_derate_factors(ocv_template_name) {
            rf_type: rise | fall | rise_and_fall;
            derate_type: early | late;
            path_type: clock | data | clock_and_data;
            index_1 ("float..., float");
            index_2 ("float..., float");
            values ( "float..., float", \
                      ...,
                      "float..., float");
        }
        ...
    }
    default_ocv_derate_group: ocv_derate_group_name;
    ocv_arc_depth: float;
    ...
    cell (cell_name) {
        ocv_derate_group: ocv_derate_group_name;
        ocv_derate(ocv_derate_group_name) {
            ocv_derate_factors(ocv_template_name) {
                rf_type: rise | fall | rise_and_fall;
                derate_type: early | late;
                path_type: clock | data | clock_and_data;
                index_1 ("float..., float");
                index_2 ("float..., float");
            }
        }
    }
}
```

```

        values ( "float,..., float", \
                  ...,\ 
                  "float,..., float");
    }
...
}
ocv_arc_depth: float;
...
pin | bus | bundle(name) {
    direction : output | inout;
    timing() {
        ocv_arc_depth: float;
        ...
    } /* end of timing */
} /* end of inout | output pin */
...
} /* end of cell */
...
} /* end of library */

```

Liberty Variation Format Parametric OCV Model Syntax

```

library (library_name) {
    lu_table_template(delay_lu_template_name) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        index_1 ("float,..., float");
        index_2 ("float,..., float");
    }
    lu_table_template(constraint_lu_template_name) {
        variable_1: input_net_transition;
        variable_2: total_output_net_capacitance;
        variable_3: related_out_total_output_net_capacitance;
        index_1 ("float,..., float");
        index_2 ("float,..., float");
        index_3 ("float,..., float");
    }
    ocv_table_template(ocv_template_name) {
        variable_1: path_distance;
        index_1 ("float,..., float");
    }
    ocv_derate(ocv_derate_group_name) {
        ocv_derate_factors(ocv_template_name) {
            rf_type: rise | fall | rise_and_fall;
            derate_type: early | late;
            path_type: clock | data | clock_and_data;
            index_1 ("float,..., float");
            values ( "float,..., float");
        }
        ...
    }
}

```

```

default_ocv_derate_distance_group: ocv_derate_group_name;
...
cell (cell_name) {
    ocv_derate_distance_group: ocv_derate_group_name;
    ocv_derate(ocv_derate_group_name) {
        ocv_derate_factors(ocv_template_name) {
            rf_type: rise | fall | rise_and_fall;
            derate_type: early | late;
            path_type: clock | data | clock_and_data;
            index_1 ("float,..., float");
            values ( "float,..., float");
        }
        ...
    }
    ...
    pin | bus | bundle (name) {
        direction : output | inout;
        timing() {
            ocv_sigma_cell_rise(delay_lu_template_name) {
                index_1 ("float,..., float");
                index_2 ("float,..., float");
                values ( "float,..., float", \
                    ...,\n
                    "float,..., float");
            }
            ocv_sigma_cell_fall(delay_lu_template_name) {
                index_1 ("float,..., float");
                index_2 ("float,..., float");
                values ( "float,..., float", \
                    ...,\n
                    "float,..., float");
            }
            ocv_sigma_rise_transition(lu_template_name) {
                sigma_type: early | late | early_and_late;
                index_1 ("float, ..., float");
                index_2 ("float, ..., float");
                index_3 ("float, ..., float");
                values ( "float, ..., float", \
                    ...,\n
                    "float, ..., float");
            }
            ocv_sigma_fall_transition(lu_template_name) {
                sigma_type: early | late | early_and_late;
                index_1 ("float, ..., float");
                index_2 ("float, ..., float");
                index_3 ("float, ..., float");
                values ( "float, ..., float", \
                    ...,\n
                    "float, ..., float");
            }
            ...
        } /* end of timing */
    } /* end of inout | output pin */
}

```

```

... ...
pin | bus | bundle (name) {
    direction : output | inout;
    timing() {
        ocv_sigma_rise_constraint(constraint_lu_template_name) {
            index_1 ("float,..., float");
            index_2 ("float,..., float");
            index_3 ("float,..., float");
            values ( "float,..., float", \
                     ...,\ \
                     "float,..., float");
        }
        ocv_sigma_fall_constraint(constraint_lu_template_name) {
            index_1 ("float,..., float");
            index_2 ("float,..., float");
            index_3 ("float,..., float");
            values ( "float,..., float", \
                     ...,\ \
                     "float,..., float");
        }
    }
    ... ...
} /* end of timing */
} /* end of inout | output pin */
...
} /* end of cell */
...
} /* end of library */

```

CCS Power Model Syntax

```

library (library_name) {
    lu_table_template (template_name) {
        variable_1: pg_voltage | pg_voltage_difference;
        index_1 ("float, ..., float");
    }
    base_curves (bc_name) {
        base_curve_type : enum (ccs_half_curve, ccs_timing_half_curve);
        curve_x ("float, ..., float");
        curve_y ("integer, float, ..., float"); /* base curve #1 */
        curve_y ("integer, float, ..., float"); /* base curve #2 */
        ...
        curve_y ("integer, float, ..., float"); /* base curve #n */
    }
    compact_lut_template (template_name) {
        base_curves_group : bc_name;
        variable_1 : input_net_transition | total_output_net_capacitance;
        variable_2 : input_net_transition | total_output_net_capacitance;
        variable_3 : curve_parameters;
        index_1 ("float, ..., float");
        index_2 ("float, ..., float");
        index_3 ("string, ..., string");
    }
}

```

```
}

pg_current_template(template_name_1) {
    variable_1 : input_net_transition;
    variable_2 : total_output_net_capacitance;
    variable_3 : time;
    index_1 ("float, ..., float");
    index_2 ("float, ..., float");
    index_3 ("float, ..., float");
cell (cell_name) {
    mode_definition (mode_name) {
        mode_value (name_string) {
            when : Boolean_expression;
            sdf_cond : Boolean_expression;
        }
    leakage_current() {
        when : Boolean_expression;
        pg_current(string) {
            value : float;
        }
        ...
        gate_leakage(string) {
            input_low_value : float;
            input_high_value : float;
        }
        ...
    }
    intrinsic_parasitic() {
        when : Boolean_expression;
        intrinsic_resistance(string) {
            related_output : string_list;
            value : float;
            reference_pg_pin : string_list;
            lut_values (template_name) {
                index_1 ("float, ..., float");
                values ("float, ..., float");
            }
        }
        intrinsic_capacitance(string) {
            value : float;
            reference_pg_pin : string_list;
            lut_values (template_name) {
                index_1 ("float, ..., float");
                values ("float, ..., float");
            }
        }
    }
}
...
dynamic_current() {
    when : Boolean_expression;
    related_inputs : string_list;
    related_outputs : string_list;
    typical_capacitances(string_list);
    switching_group() {
```

```

    input_switching_condition(string);
    output_switching_condition(string_list);
    pg_current(string) {
        vector(string) {
            reference_time : float;
            index_output(string);
            index_1(float);
            ...
            index_n(float);
            index_n+1(string_list);
            values(float, ..., float);
        }
        compact_ccs_power (template_name) {
            base_curves_group : bc_name;
            index_output : pin_name;
            index_1 ("float, ..., float");
            index_2 ("float, ..., float");
            index_3 ("string, ..., string");
            values ("float | integer, ..., float | integer");
        }
    }
}
...
}
} /* end of cell */
...
} /* end of library */

```

Note:

The CCS power model can either be compact or noncompact, but not both.

Power-Aware Model Syntax

Level-Shifter Cell

```

library(library_name) {
    cell(cell_name) {
        is_level_shifter : boolean ;
        level_shifter_type : enum ;
        input_voltage_range ("float, float");
        output_voltage_range ("float, float");
        pin | bus | bundle (pin_name) {
            input_voltage_range ("float, float");
            output_voltage_range ("float, float");
            level_shifter_data_pin : boolean ;
            level_shifter_enable_pin : boolean ;
        }/* end pin group*/
    }/* end cell group*/
}/* end library group*/

```

Isolation Cell

```
library(library_name) {
    cell(cell_name) {
        is_isolation_cell : boolean ;
        is_clock_isolation_cell : boolean ;
        pin | bus | bundle (pin_name) {
            always_on : Boolean ;
            isolation_cell_data_pin : boolean ;
            isolation_cell_enable_pin : boolean ;
            clock_isolation_cell_clock_pin : boolean ;
            alive_during_partial_power_down : boolean ;
        }/* end pin group*/
    }/* end cell group*/
}/* end library group*/
```

Retention Cell

```
library(library_name) {
    cell(cell_name) {
        retention_cell : retention_cell_style;
        retention_condition() {
            power_down_function: "Boolean_function";
            required_condition: "Boolean_function";
        }
        clock_condition() {
            clocked_on : string;
            required_condition : string;
            hold_state : enum ;
            clocked_on_also : string;
            required_condition_also : string;
            hold_state_also : enum;
        }
        preset_condition() {
            input : string;
            required_condition : string ;
        }
        clear_condition() {
            input : string ;
            required_condition : string ;
        }
        ff (["reference_pin_names",] variable1, variable2 ) {
            power_down_function : string ;
        }
        latch (["reference_pin_names",] variable1, variable2 ) {
            power_down_function : string;
        }
        ff_bank (["reference_pin_names",] variable1, variable2, bits) {
            power_down_function : string;
        }
        latch_bank (["reference_pin_names",] variable1, variable2, bits) {
            power_down_function : string;
        }
        statetable (...) {
```

```

        power_down_function : string;
    }
    pin | bus | bundle (pin_name) {
        always_on : Boolean ;
        retention_pin(pin_class, disable_value);
        save_action : L|H|R|F;
        restore_action : L|H|R|F;
        restore_edge_type : edge_trigger | leading | trailing;
    }/* end pin group*/
}/* end cell group*/
}/* end library group*/

```

Switch Cell

```

library(library_name) {
    cell(cell_name) {
        switch_cell_type : enum;
        pin | bus | bundle (pin_name) {
            switch_pin : Boolean ;
            always_on : Boolean ;
        }/* end pin group*/
    }/* end cell group*/
}/* end library group*/

```

Always-On Cell

```

library(library_name) {
    cell(cell_name) {
        always_on : Boolean ;
        pin | bus | bundle (pin_name) {
            always_on : Boolean ;
            is_isolated : boolean;
            isolation_enable_condition : string;
        }/* end pin group*/
    }/* end cell group*/
}/* end library group*/

```

PG Pin

```

library(library_name) {
    define(); // for user-defined attributes
    define_group(); // for user-defined groups
    cell(cell_name) {
        pg_pin (pg_pin_name) {
            direction : enum;
            permit_power_down : boolean;
            pg_function : string;
            pg_type : enum;
            physical_connection : enum;
            related_bias_pin : string;
            std_cell_main_rail : boolean;
            switch_function : string;
            user_pg_type : string;
            voltage_name : string; // do not allow update
        }
    }
}

```

```

        is_insulated      : boolean;
        tied_to          : string;
    }
pin/bus/bundle (pin_name) {
    related_power_pin : string; // generic PG attribute
    related_ground_pin : string; // generic PG attribute
    related_bias_pin : string; // generic PG attribute
    power_down_function : string; // generic PG attribute
}/* end pin group*/
}/* end cell group*/
}/* end library group*/

```

Reference Syntax

This section lists the model file reference syntax.

General

This section lists the reference syntax applicable to all the model types.

Library Units

Different model files require different library units.

```

library(library_name) {
    distance_unit: float; //required by aocv and pocv model files
    time_unit: float; //required by aocv, pocv and ccs_power model files
    capacitive_load_unit(float, string);
    // required by aocv, pocv and ccs_power model files
    pulling_resistance_unit: float; // required by ccs_power model files
    voltage_unit: float; // required by ccs_power model files
    current_unit: float; // required by ccs_power model files
}

```

Operating Conditions

Required for all Liberty model files. Screener rules are the same as base library.

```

library(library_name) {
    nom_process: float;
    nom_voltage: float;
    nom_temperature: float;
    operating_conditions() {
        ...
    }
    default_operating_conditions: string;
    ...
}

```

Bus-Related Definitions

If the model file contains bus syntax, bus-related syntax definition is required as the .db file needs to be bit-blasted. This is required for all Liberty model files. Screener rules are the same as base library.

```
library(library_name) {
    bus_naming_stype: string;
    type (string) {
        base_type : string;
        data_type : string;
        bit_width : int;
        bit_from : int;
        bit_to : int;
    }
    cell(cell_name) {
        bus(bus_name) {
            bus_type : string;
        }
    }
}
```

Delay Model Definition

The `delay_model` attribute is required for lookup table definition. This is required for all model files. Screener rules are the same as base library.

```
library() {
    delay_model: model_name;
    ...
}
```

Reference Syntax For Specific Model Types

This section lists the reference syntax applicable only to the specific model types.

Timing Definition for LVF Parametric OCV Model file

If the model file contains timing group syntax, attributes to identify the timing group are included. This is required for the Liberty variation format (LVF) parametric OCV model file. Screener rules are the same as base library.

```
timing() {
    related_pin: string;
    timing_type: string;
    timing_sense: string;
    when: Boolean_expression;
    ...
}
```

PG Pin Definition for CCS Power Model file

The `pg_pin` group is referenced by CCS power syntax. This is required for the CCS power model file. Screener rules are the same as base library.

```
library(library_name) {
    voltage_map(string, float);
    cell(cell_name) {
        pg_pin(pg_pin_name) {
            voltage_name : string;
            pg_type : string;
            user_pg_type : string;
        }
    }
}
```

Reference Syntax For Power-Aware Model Files

Signal Pin Definition

Optional. The `pin` group is referenced by power-aware syntax. This is required only to check the consistency and correctness of the power management attributes. Screener rules are the same as base library.

```
pin | bus | bundle (pin_name) {
    direction : enum;
}
```

PG Pin Definition

Optional. This is required only to check the consistency and correctness of the power management attributes.

```
library(name) {
    voltage_map(voltage_name, dummy_value);
    cell(cell_name) {
        pg_pin(pg_pin_name) {
            direction : enum;
            permit_power_down : boolean;
            pg_function : string;
            pg_type : enum;
            physical_connection : enum;
            related_bias_pin : string;
            std_cell_main_rail : boolean;
            switch_function : string;
            user_pg_type : string;
            voltage_name : string;
            is_insulated : boolean;
            tied_to : string;
        }
    }
}
```

Miscellaneous Attributes

Optional. This is required only to turn on the macro cell specific screener checks.

```
cell(cell_name) {
    is_macro_cell : boolean;
}
```

Framework Syntax For Power-Aware Model Files

The cell and pin names must exist in the base library. The `ff`, `latch`, `ff_bank`, `latch_bank`, and `statetable` group definitions must be complete in the base library and the corresponding group headers must be identical in the model file.

```
library(library_name) {
    cell(cell_name) {
        ff (["reference_pin_names"],] variable1, variable2 ) {
        }
        latch (["reference_pin_names"],] variable1, variable2 ) {
        }
        ff_bank (["reference_pin_names"],] variable1, variable2, bits) {
        }
        latch_bank (["reference_pin_names"],] variable1, variable2, bits) {
        }
        statetable (...) {
        }
        pin | bus | bundle (pin_name) {
        }/* end pin group*/
        pg_pin (pg_pin_name) {
        }/* end pg_pin group*/
    }/* end cell group*/
}/* end library group*/
```

The `pg_pin` group can be a part of the model, reference, or framework syntax:

- If the `pg_pin` group in the model file is used only to update the corresponding base library attribute values, it is considered to be part of the framework syntax and must exist in the base library.
- If the `pg_pin` group in the model file is used to check the other model file syntax, it is considered to be part of the reference syntax. The group header and the `pg_type` attribute value must be identical to that in the base library.
- If the `pg_pin` group is a new group to be added to the base library, it is considered to be part of the model syntax and must not exist in the base library.

Compiling Model Files

Use the `-model_type` option with the `read_lib` command to compile the model files, as shown:

```
prompt> read_lib -model_type {list_of_model_types} file_name
```

where `file_name` specifies the input .lib model file.

The `list_of_model_types` argument specifies a list of the model types. The valid values are `lvf`, `ccs_power`, and `power_aware`.

The model file syntax is compiled based on the specified model types. For example,

- The `read_lib` command compiles only the power-aware syntax when you specify the `-model_type power_aware` option:

```
prompt> read_lib -model_type power_aware upf_model_file.lib
```

- The `read_lib` command compiles only the Liberty variation format (LVF) syntax when you specify the `-model_type lvf` option:

```
prompt> read_lib -model_type lvf lvf_model_file.lib
```

- The `read_lib` command compiles only the CCS power syntax when you specify the `-model_type ccs_power` option:

```
prompt> read_lib -model_type ccs_power ccsp_model_file.lib
```

- The `read_lib` command compiles both the Liberty variation format and CCS power syntax when you specify the `-model_type {lvf ccs_power}` option:

```
prompt> read_lib -model_type {lvf ccs_power} lvf_ccsp_model_file.lib
```

The Library Compiler tool also checks the model file syntax based on the specified model type. For example, if you specify the `-model_type lvf` option with the `read_lib` command and the input model file library contains the CCS Power `dynamic_current` group, the tool generates an error message.

The `read_lib` command checks the screener rules only for the Liberty model files specified with the `-model_type` option. The command skips the screener rules that check the relationship between the model file and the base library. For example, if the specified Liberty model files type is `lvf`, the `read_lib` command does not check for the nominal timing table syntax.

Table 19-2 shows some examples of the screener rules skipped by the `read_lib` command for different Liberty model files.

Table 19-2 Screener Rules Skipped by the read_lib Command for Different Liberty Model Files

Model file type	Error Id	Description
lvf	LBDB-267	Checks for existence of required nominal timing table
	LBDB-559	Checks for same indexes between sigma and nominal table
ccs_power	LBDB-607	Checks for existence of required internal power when using new power and ground pin syntax

Compiling Power-Aware Model Files

To compile a power-aware model file, use the `read_lib -model_type` command with the `power_aware` argument as shown:

```
prompt> read_lib -model_type power_aware upf_model_file.lib
```

When you use the `-model_type power_aware` option, the `read_lib` command checks

- The syntax of individual attributes, such as the lower bound must not exceed the upper bound in the `input_voltage_range` definition
- The syntax of multiple attributes including consistency and conflict checks, such as the `input_voltage_range` attribute must be defined only on an input pin and not on an output pin

If you use the model file only to add or update a few attributes in the base library, you do not need to check for consistency and conflict across multiple attributes. In such cases, use the `-minimum_check` option with the `-model_type power_aware` option as shown:

```
lc_shell> read_lib model_file_name -model_type power_aware -minimum_check
```

When you specify the `-minimum_check` option, the `read_lib` command checks only the syntax of individual attributes. You use this option only with the `-model_type power_aware` option.

For example, the following model file is used to update the `output_voltage_range` attribute values in the base library. When you compile this file using the `-minimum_check` option, the tool checks only the `output_voltage_range` attribute and the model file is compiled.

```
library(empty) {
```

```

cell(cell1) {
    pin(pin1) {
        output_voltage_range ("0.7, 1.2");
    }
}
}

```

If you do not use the `-minimum_check` option, the compilation fails with the following error:

Error: Line 4, Cell 'cell1', pin 'pin1', The 'output_voltage_range' attribute cannot be specified on the pin 'pin1'.

This is because the tool checks the attribute against the cell type, which is not level-shifter.

To compile this file without using the `-minimum_check` option, add the cell type and other necessary level-shifter attributes to the model file as shown:

```

library(empty) {
    cell(cell1) {
        is_level_shifter : true;
        pg_pin(VSS) {
            voltage_name : VSS;
            pg_type : primary_ground;
        }
        pg_pin(VDD) {
            voltage_name : VDD;
            pg_type : primary_power;
        }
        pg_pin(VCC) {
            voltage_name : VCC;
            pg_type : primary_power;
        }
        pin(input) {
            related_power_pin : VCC;
            related_ground_pin : VSS;
            direction: input;
            input_voltage_range ("3.7, 3.9");
        }
        pin(pin1) {
            related_power_pin : VDD;
            related_ground_pin : VSS;
            direction: output;
            output_voltage_range ("0.7, 1.2");
        }
    }
}

```

20

Modeling Timing of Complex Macro Blocks

This chapter describes the procedure and methodology for specifying the static timing of complex macro blocks used as black boxes.

The interface timing specification applies to ASIC libraries containing sequential blocks modeled as black boxes whose functionality is either too complex to model or not required.

These are the timing concepts described in this chapter:

- [Interface Timing Specification](#)
- [Describing Blocks With Interface Timing](#)
- [Using Blocks With Interface Timing in Synthesis](#)
- [Interpreting Timing Relationships](#)
- [Examples Using Interface Timing Specification](#)

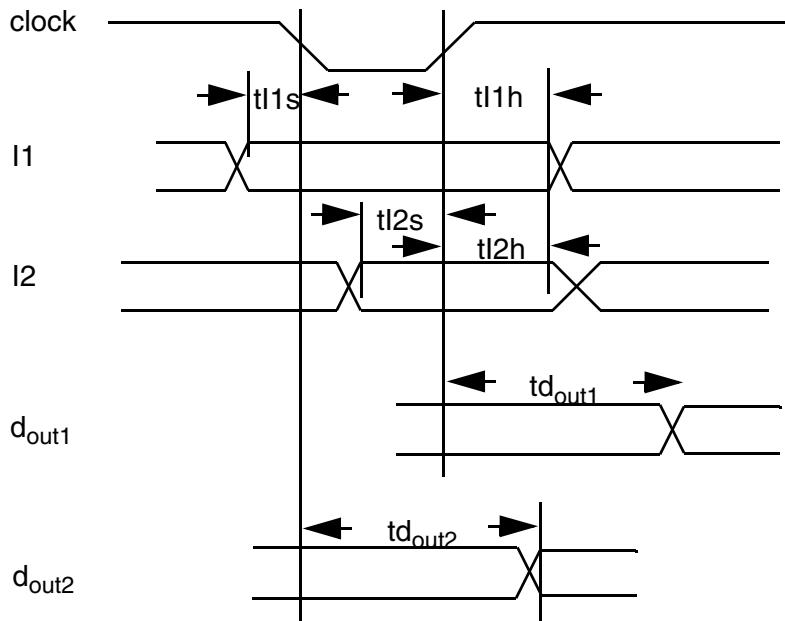
Interface Timing Specification

The interface timing specification of a sequential block describes the static timing of a block as seen from the outside. This specification does not include the timing of the internal paths of the block.

A complex sequential block contains more than one sequential device. It can have more than one driving clock to which you specify input constraints and output delays. Such blocks include RAMs and hard macros, such as the test access port (TAP) controller circuit in the Joint Test Action Group (JTAG).

[Figure 20-1](#) is an example of the interface timing of a complex block.

Figure 20-1 Interface Timing of a Complex Block



In [Figure 20-1](#),

- Input I1 has a setup constraint with respect to the falling edge of the clock and a hold constraint with respect to the rising edge of the clock.
- The setup constraints for inputs I1 and I2 are specified with respect to different edges of the clock.
- Output data d_{out1} is propagated on the rising edge of the clock. Output data d_{out2} is propagated on the falling edge of the clock.

Sequential Cell Timing Without Interface Timing Specification

In static timing, a sequential cell is considered to contain one or more edge-triggered flip-flops (rising or falling edge) or level-sensitive latches (negative or positive level). When functional information is not available during synthesis, the interpretation of the cell is based on the various timing relationships specified in the cell description. Cells are interpreted as follows:

- As a simple edge-triggered device if the setup arc and the output delay arc are specified for the same clock edge
- As a master-slave device if the setup arc and the output delay arc are specified for different edges of the clock
- As a level-sensitive device if it has at least one output that combines a combinational arc from a data input and a sequential arc from a clock. However, if cells do not have a sequential arc from the clock to the output port with a combinational arc from the D pin to the output port, then an edge-triggered register is inferred.

When the cell is interpreted as a simple edge-triggered, master-slave, or level-sensitive device, some of the timing information specified in the library is ignored. For example, if a cell is interpreted as a rising-edge-triggered device, the Library Compiler tool checks the hold constraint on the rising edge even if this constraint is specified for the falling edge of the clock.

Supported Features of Interface Timing

Although the rules described in the previous section work well for simple sequential cells, they do not apply to complex sequential blocks. The interface timing capability supports the timing of complex sequential blocks without any assumptions about their underlying structure.

During synthesis, timing relationships are interpreted according to the following well-defined set of rules, in which each relationship is interpreted independently of other relationships:

- Interface timing supports multiple clock signals. A block can have one or more clock inputs with different waveforms for each clock.
- Setup and hold constraints of data inputs can be specified with respect to the rising or falling edges of one or more clocks.

A control input can have recovery or removal arcs specified with respect to one or more clocks. Note that recovery and removal constraints are not checked by synthesis.

- Propagation delays can be specified with respect to the rising or falling edges of one or more clocks.

An output pin can have a propagation delay specified with respect to the falling or rising edge of any clock. One output can have several propagation delays specified with respect to several clock inputs.

- Combinational delay can be specified with respect to one or more data inputs.
An output pin can also have a delay arc from one or more inputs. This arc can coexist with other arcs from clock inputs and can be a regular delay arc, an asynchronous reset, or a clear arc. Regular delay arcs include combinational arcs.
- Timing properties of clocks, such as minimum period and minimum pulse width, can be specified in the same way that they are specified for regular library cells. These constraints are not checked by synthesis.

The benefits of using the interface timing capability to describe the static timing of complex sequential blocks are:

- Higher-quality results from synthesis

With accurate timing models for complex blocks, optimization of the logic around these blocks is better, because the synthesis process does not miss real violations or detect false violations. When the synthesis process detects false violations, it optimizes the wrong path, making the circuit larger and faster than it needs to be. Real violations missed in synthesis, which are detected in simulation after synthesis, require more synthesize-simulate iterations and, consequently, a longer development cycle.

- Support for complex sequential blocks

The timing of complex sequential blocks can be specified for the synthesis tools to use when making optimization decisions.

- Abstraction and information hiding

Because interface timing describes the static timing of a block as seen from the outside, it is not necessary to specify the internal structure of the block or the timing paths internal to the block, which are assumed to have been verified when the block was built.

For example, if the block has a corresponding gate-level netlist, it is not necessary to export the netlist to be able to time through it. You can use interface timing specification to specify the static timing of the block while keeping it as a black box in terms of internal structure and functionality. You instantiate the block in the design, compile the design, and export it to downstream tools.

During all the phases of the synthesis process, the block is treated as a leaf cell in the design. Functional information is needed only when the design is being verified through simulation when a functional simulation model of the block can be used.

Describing Blocks With Interface Timing

This section describes how to use Library Compiler syntax to specify the interface timing of complex blocks. The following sections describe the design flow of these blocks through synthesis.

Using Library Compiler Syntax

To specify the interface timing of a macro block, describe the block as a cell or model in the library and use `timing group` syntax to describe the timing arcs. The Library Compiler tool treats this `cell or model` group like a regular `cell or model` group in an ASIC library.

Differences From Regular Cell Specification

These are the differences from regular cell description for specifying the interface timing of complex sequential blocks in Library Compiler syntax:

1. Add this attribute and value to the `cell` group:

```
interface_timing : true;
```

This indicates that the timing arcs must be interpreted according to interface timing specification semantics. If this attribute is missing or its value is false, the timing relationships are interpreted as those of a regular cell rather than according to interface timing specification semantics.

2. Cells with interface timing are strictly black boxes, in terms of functionality. In other words, the `function` attribute must not be specified for any output pin of the cell. In addition, the `cell` group cannot contain an `ff` group. As a consequence, cells with interface timing cannot be inferred but must be instantiated.

Advanced Modeling Technology

To work effectively with the PrimeTime tool to handle chip-level designs, Synopsys expanded the Library Compiler interface timing specification modeling capability to support the PrimeTime modeling requirement.

For more information about the PrimeTime tool, see the *PrimeTime User Guide*.

The next two sections describe new attributes related to PrimeTime support.

model Group

A `model` group describes limited hierarchical interconnections. Currently, a `model` group is required only when shorted ports are present. When a `model` group is used, the Library Compiler tool writes out a separate model design .db as an instantiation of the model cell plus the net connections for the shorted ports.

The Library Compiler tool generates hierarchical .db designs with the `model` group, which can accept all the groups and attributes that a `cell` group accepts, plus these additional two new attributes:

- `cell_name` simple attribute
- `short` complex attribute

Syntax

```
model(model_name) {
    area : float;
    ...
    cell_name : cell_string; /*optional*/
    ...
    pin(A, Y) { ... }
    short (A, Y);
}
```

cell_name Simple Attribute

The `cell_name` attribute specifies the name of the cell within a `model` group.

Syntax

```
cell_name : name_id ;
```

Example

```
model(modelA) {
    cell_name : "cellA";
    ...
}
```

short Complex Attribute

The `short` complex attribute lists the shorted ports that are connected by metal or polytrace. These ports are modeled within a `model` group.

The most common example of a shorted port is a feedthrough, where an input port is directly connected to an output port. Another example is two output ports that fan out from the same gate.

Syntax

```
short("name_list") ;
```

Example

```
short (b, y);
```

model Group Example

[Example 20-1](#) shows a `model` group with `short` attributes.

Example 20-1 model Group With short Attributes

```
library(TEST) {
    model(modelA) {
        area : 0.4;
        short(b, y);
        short(c, y);
        cell_name : "cellA";
        pin(y) {
            direction : output;
            timing() {
                related_pin : a;
            }
        }
        pin(a) {
            direction : input;
            capacitance : 0.1;
        }
        pin(b) {
            direction : input;
            capacitance : 0.1;
        }
        pin(c) {
            direction : input;
            capacitance : 0.1;
            clock : true;
        }
    }
}
```

generated_clock Group

A `generated_clock` group is defined within a `cell` group or a `model` group to describe a new clock that is generated from a master clock by

- clock frequency division
- clock frequency multiplication
- edge derivation

Syntax

```
cell (name_id) {
    ...generated_clock description...
}
```

Simple Attributes

clock_pin
divided_by
duty_cycle
invert
master_pin
multiplied_by

Complex Attributes

edges
shifts

clock_pin Simple Attribute

Required in the *generated_clock* group. The *clock_pin* attribute identifies a pin connected to an input clock signal.

Syntax

```
clock_pin : name ;
```

Example

```
clock_pin : clk1;
```

divided_by Simple Attribute

The *divided_by* attribute specifies the frequency division factor, which must be a power of 2.

Syntax

```
divided_by : integer;
```

Example

```
generated_clock(genclk1) {
    clock_pin : clk1;
    master_pin : clk;
    divided_by : 2;
}
```

This code fragment shows a clock pin (clk1) generated by division of the original clock pin (clk) frequency by 2.

duty_cycle Simple Attribute

The `duty_cycle` attribute specifies the duty cycle (expressed as a floating-point number) for frequency multiplied clocks. Use this option for high pulse width.

Syntax

```
duty_cycle : float;
```

Example

```
generated_clock(genclk2) {
    clock_pin : clk1;
    master_pin : clk;
    multiplied_by : 2;
    duty_cycle : 50.0;
}
```

This code fragment shows a clock pin (clk1) generated by multiplication of the original clock pin (clk) frequency by 2 with a duty cycle of 50.

invert Simple Attribute

The `invert` attribute inverts the waveform generated by multiplication or division. Set this attribute to true to invert the waveform. Set it to false (default) if you do not want to invert the waveform.

Syntax

```
invert : true | false ;
```

Example

```
generated_clock(genclk1) {
    clock_pin : clk1;
    master_pin : clk;
    divided_by : 2;
    invert : true;
}
```

This code fragment shows a clock pin (clk1) generated by division of the original clock pin (clk) frequency by 2 and then inversion.

master_pin Simple Attribute

Required in the `generated_clock` group. The `master_pin` attribute identifies the master (original) clock pin.

Syntax

```
master_pin : name ;
```

Example

```
master_pin : clk;
```

multiplied_by Simple Attribute

The `multiplied_by` attribute specifies the frequency multiplication factor, which must be a power of 2.

Syntax

```
multiplied_by : integer;
```

Example

```
generated_clock(genclk2) {
    clock_pin : clk1;
    multiplied_by : 2;
    master_pin : clk;
}
```

This code fragment shows a clock pin (clk1) generated by multiplication of the original clock pin (clk) frequency by 2.

Syntax

```
edges (edge1,edge2,edge3);
```

Example

```
edges (1, 3, 5);
```

edges Complex Attribute

The `edges` attribute specifies a list of edges (in terms of the edge numbers) from the master clock that form the edges of the generated clock. You must specify three edges. Use this option when simple division or multiplication is insufficient to describe the generated clock waveform.

shifts Complex Attribute

The `shifts` attribute specifies the shifts (in time units) to be added to the edges specified in the edge list to generate the clock. The number of shifts must equal the number of edges (that is, three). This shift modifies the ideal clock edges (it is not considered as clock latency).

Syntax

```
shifts (shift1,shift2,shift3);
```

Example

```
shifts (5.0, -5.0, 0.0);
```

Generated Clock Example

[Example 20-2](#) shows a generated clock description.

Example 20-2 Generated Clock Description

```
cell(acell) {
    ...
    generated_clock(genclk1) {
        clock_pin : clk1;
        master_pin : clk;
        divided_by : 2;
        invert : true;
    }
    generated_clock(genclk2) {
        clock_pin : clk1;
        master_pin : clk;
        multiplied_by : 2;
        duty_cycle : 50.0;
    }
    generated_clock(genclk3) {
        clock_pin : clk1;
        master_pin : clk;
        edges(1, 3, 5);
        shifts(5.0, -5.0, 0.0);
    }
    ...
    pin(clk) {
        direction : input;
        clock : true;
        capacitance : 0.1;
    }
    pin(clk1) {
        direction : input;
        clock : true;
        capacitance : 0.1;
    }
}
```

Similarities to Regular Cell Specification

These are the ways interface timing specification is similar to regular cell description:

1. Associate the block with a library. Use one of these two methods:
 - Specify the `cell` group or the `model` group within a `library` group and use the `read_lib` command to import the library into .db format.
2. Use the same delay model for interface timing of a block as the one defined in its associated library. For example, associate the block using the linear delay model to specify timing arcs with a library using the linear model. Similarly, associate the block using the nonlinear delay model with a library using the nonlinear model.

3. Use the same timing arc syntax for describing regular cells to specify the timing arcs that describe the interface timing relationships.

According to the interface timing specification policy, blocks with interface timing are always considered black boxes in terms of functionality. To simulate designs containing blocks with interface timing, you must provide a functional simulation model of such blocks.

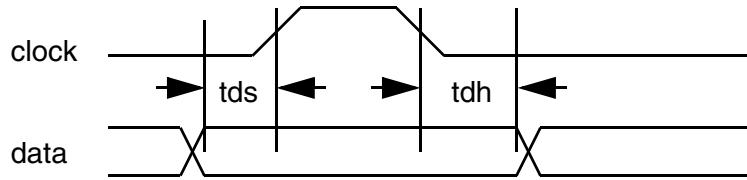
Examples of Interface Timing Relationships

This section describes the various timing relationships supported by the interface timing specification and provides an example of a fairly complex sequential block with two clocks, four data inputs, and two data outputs.

Input Setup and Hold

A data input can have a setup or hold constraint specified with respect to the rising or falling edge of any clock input. [Figure 20-2](#) shows a data input with the setup constraint specified with respect to the rising edge of the clock and the hold constraint specified with respect to the falling edge of the clock.

Figure 20-2 Data Input With Specified Setup and Hold Constraints



[Example 20-3](#) shows the Library Compiler description of these relationships.

Example 20-3 Description of Timing Relationships in Figure 20-2

```
pin (I2) {
    timing () /* tds */
        timing_type : setup_rising;
        cell_rise(scalar) {values( " 2.20 ");}
        cell_fall(scalar) {values( " 2.20 ");}
        related_pin : "clock";
    }
    timing () /* tdh */
        timing_type : hold_falling;
        cell_rise(scalar) {values( " 0.95 ");}
        cell_fall(scalar) {values( " 0.95 ");}
        related_pin : "clock";
}
```

Output Propagation

The output delay can be specified with respect to the rising or falling edge of the clock. Figure 20-3 shows the output section of a block. Output O1 is propagated on the rising edge of the clock, and output O2 is propagated on the falling edge.

Figure 20-3 Output Section of a Macro Block

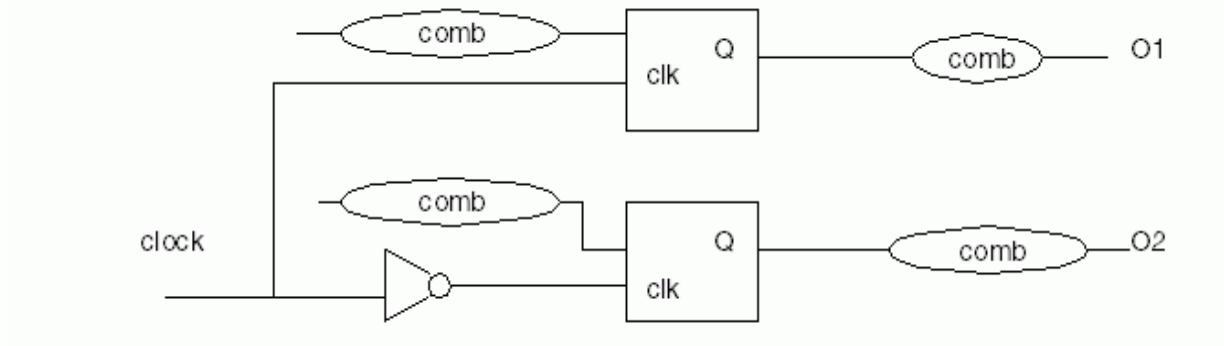
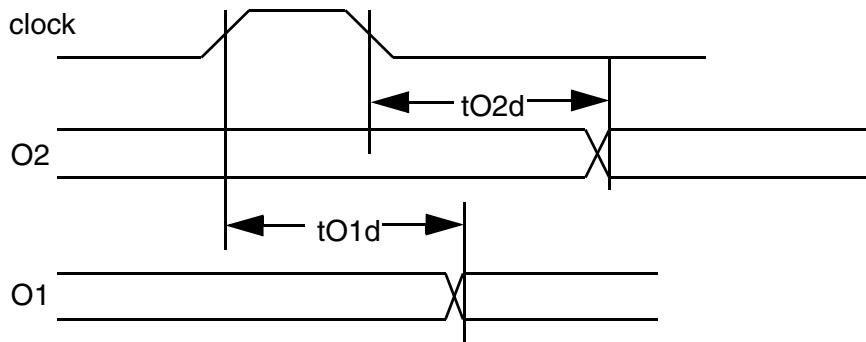


Figure 20-4 shows the interface timing relationships for the two outputs.

Figure 20-4 Output Timing Relationships for the Block in Figure 20-3



Example 20-4 shows the Library Compiler description of the relationships in Figure 20-4.

Example 20-4 Description of Timing Relationships in Figure 20-4

```

pin(O1) {
    direction : output;
    timing () { /* tO1d
        timing_type : rising_edge; /*sequential delay arc*/
        cell_rise(scalar) {values( " 2.25 ");}
        cell_fall(scalar) {values( " 2.57 ");}
        rise_transition(scalar) {values( " 0.020 ");}
        fall_transition(scalar) {values( " 0.017 ");}
        related_pin : "clock";
    }
}

```

```

        }

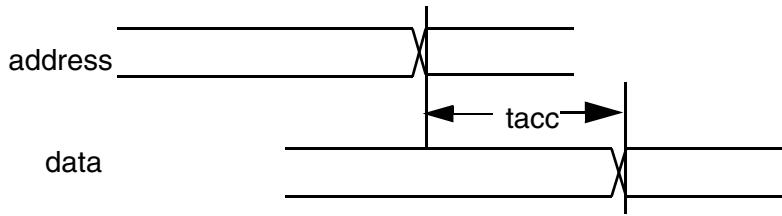
pin(02) {
    direction : output;
    timing () /* t02d */
        timing_type : falling_edge; /*sequential delay arc*/
        cell_rise(scalar) {values( " 3.15 ");}
        cell_fall(scalar) {values( " 3.00 ");}
        rise_transition(scalar) {values( " 0.020 ");}
        fall_transition(scalar) {values( " 0.017 ");}
        related_pin : "clock";
    }
}

```

Through Combinational Paths

An output of a sequential block can have a propagation delay specified with respect to a nonclock input. This relationship indicates the existence of a direct combinational path between the output and the input. The output always changes (after some delay) when the input changes. [Figure 20-5](#) shows such a case.

Figure 20-5 Timing Relationships for Through Combinational Paths



[Example 20-5](#) shows the Library Compiler description of the relationships in [Figure 20-5](#).

Example 20-5 Description of Timing Relationships in Figure 20-5

```

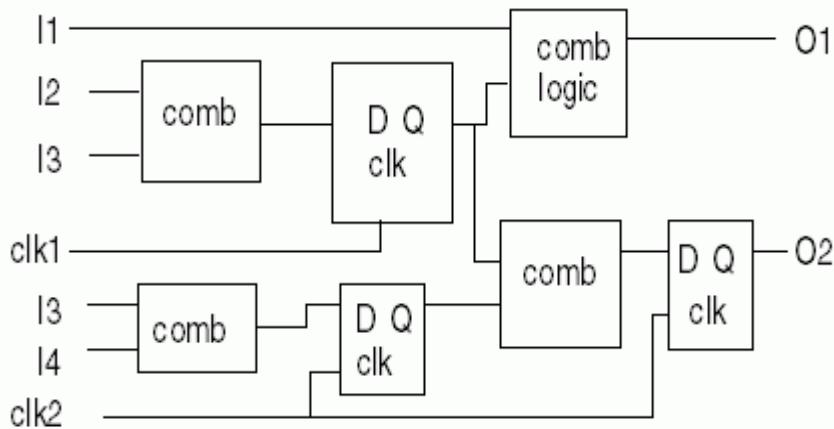
bus(address) {
    direction : input;
    bus_type   : "bus10"
}
bus(data){
    direction : output;
    bus_type   : "bus8"
    timing () {
        timing_sense : non_unate;
        cell_rise(scalar) {values( " 8.15 ");}
        cell_fall(scalar) {values( " 7.00 ");}
        rise_transition(scalar) {values( " 0.020 ");}
        fall_transition(scalar) {values( " 0.017 ");}
        related_pin : "address";
    }
}

```

Interface Timing of a Complex Sequential Block

[Figure 20-6](#) shows the internal structure of a complex block containing sequential and combinational logic. Note that the input constraints for the sequential elements in the circuit are specified with respect to both edges of the clock.

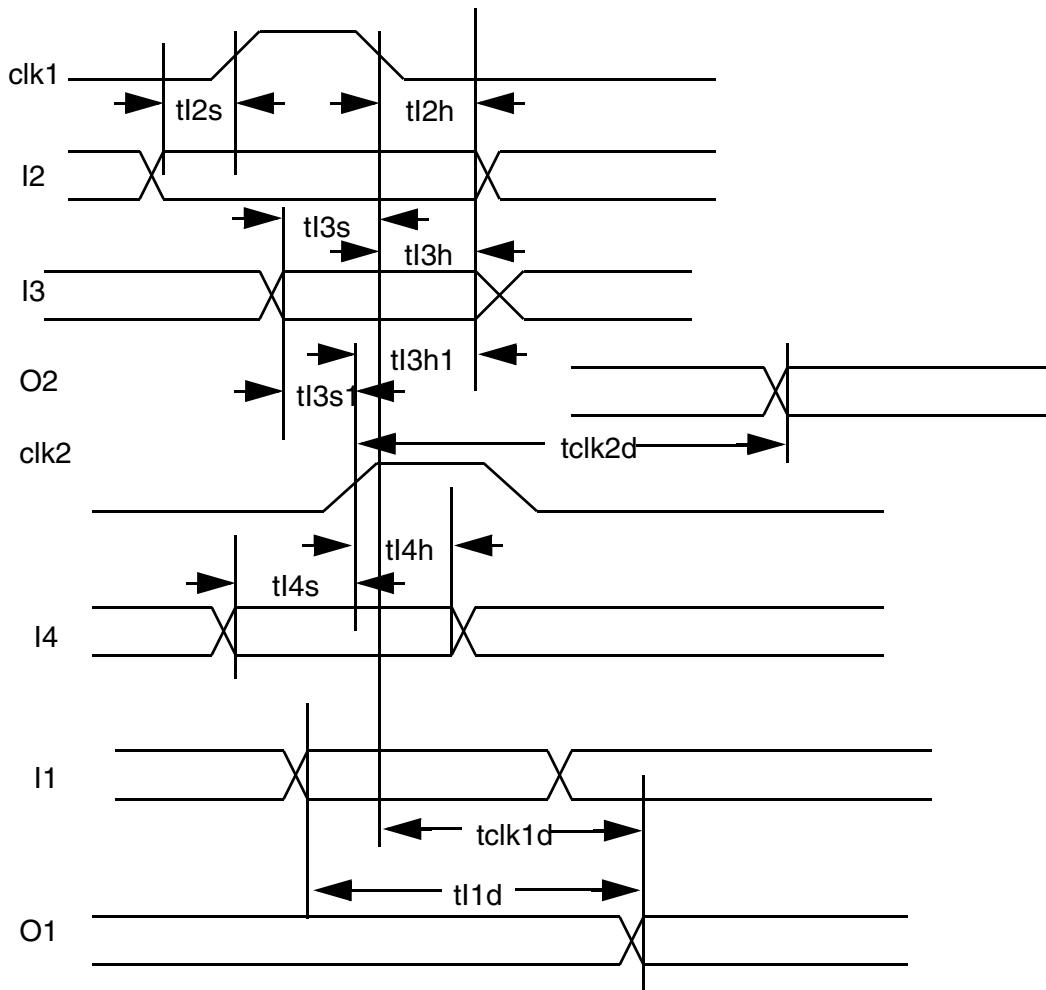
Figure 20-6 Complex Sequential Block Supported by the Interface Timing Specification



[Figure 20-7](#) is the timing diagram illustrating the various interface relationships of this block.

- The block has two clock pins, clk1 and clk2.
- Input I2 has a setup constraint with respect to the rising edge of clk1 and a hold constraint with respect to the falling edge of the same clock.
- Input I3 has setup and hold constraints specified with respect to both clocks (rising for clk2 and falling for clk1).
- Output O1 is propagated on the falling edge of clk1, and output O2 is propagated on the rising edge of clk2.
- Output O1 combines a combinational path from I1 and a sequential path from clk1.

Figure 20-7 Timing Relationships for the Circuit in Figure 20-6



[Example 20-6](#) shows the library description of the interface timing specification of the block shown in [Figure 20-6](#). The CMOS delay model is used for this cell.

Example 20-6 Description of the Timing Relationships in Figure 20-6

```
cell (XR_2120) {
    area : 100.000000;
    pin (I1) {
        direction : input;
        capacitance : 1.46;
        fanout_load : 1.46;
    }
    pin (I2) {
        direction : input;
        capacitance : 1.46;
        fanout_load : 1.46;
        timing () { /* tI2s */
```

```
timing_type : setup_rising;
rise_constraint(scalar) {
    values( " 2.20 ");
}
fall_constraint(scalar) {
    values( " 2.20 ");
}
related_pin : "clk1";
}
timing () /* tI2h */
timing_type : hold_falling;
rise_constraint(scalar) {
    values( " 0.95 ");
}
fall_constraint(scalar) {
    values( " 0.95 ");
}
related_pin : "clk1";
}
pin (I3) {
    direction : input;
    capacitance : 1.46;
    fanout_load : 1.46;
    timing () /* tI3s */
        timing_type : setup_falling;
        rise_constraint(scalar) {
            values( " 1.20 ");
        }
        fall_constraint(scalar) {
            values( " 1.20 ");
        }
        related_pin : "clk1";
}
timing () /* tI3h */
timing_type : hold_falling;
rise_constraint(scalar) {
    values( " 1.95 ");
}
fall_constraint(scalar) {
    values( " 1.95 ");
}
related_pin : "clk1";
}
timing () /* tI3s1 */
timing_type : setup_rising;
rise_constraint(scalar) {
    values( " 0.70 ");
}
fall_constraint(scalar) {
    values( " 0.70 ");
}
related_pin : "clk2";
```

```
        }
    timing () /* tI3h1 */
        timing_type : hold_rising;
        rise_constraint(scalar) {
            values( " 2.95 ");
        }
        fall_constraint(scalar) {
            values( " 2.95 ");
        }
        related_pin : "clk2";
    }

}
pin (I4) {
    direction : input;
    capacitance : 1.46;
    fanout_load : 1.46;
    timing () /* tI4s */
        timing_type : setup_rising;
        rise_constraint(scalar) {
            values( " 1.70 ");
        }
        fall_constraint(scalar) {
            values( " 1.70 ");
        }
        related_pin : "clk2";
    }
    timing () /* tI4h */
        timing_type : hold_rising;
        rise_constraint(scalar) {
            values( " 0.80 ");
        }
        fall_constraint(scalar) {
            values( " 0.80 ");
        }
        related_pin : "clk2";
    }
}
pin (clk1) {
    direction : input;
    capacitance : 1.13;
    fanout_load : 1.13;
    clock : true;
    /* clock pins must be labeled as such */
}
pin (clk2) {
    direction : input;
    capacitance : 1.13;
    fanout_load : 1.13;
    clock : true;
    /* clock pins must be labeled as such */
}
pin (O1){
```

```
direction : output;
timing () /* tI1d */
    timing_sense : non_unate; /* combinational delay */
    cell_rise(scalar) {
        values( " 3.25 ");
    }
    rise_transition(scalar) {
        values( " 0.020 ");
    }
    cell_fall(scalar) {
        values( " 3.50 ");
    }
    fall_transition(scalar) {
        values( " 0.017 ");
    }
    related_pin : "I1";
}
timing () /* tclk1d */
    timing_type : falling_edge;
/* sequential delay arc */
    cell_rise(scalar) {
        values( " 1.25 ");
    }
    rise_transition(scalar) {
        values( " 0.020 ");
    }
    cell_fall(scalar) {
        values( " 1.50 ");
    }
    fall_transition(scalar) {
        values( " 0.017 ");
    }
    related_pin : "clk1";
}
pin (O2){
    direction : output;
    timing () /* tclk2d */
        timing_type : rising_edge;
/*sequential delay arc */
        cell_rise(scalar) {
            values( " 2.25 ");
        }
        rise_transition(scalar) {
            values( " 0.020 ");
        }
        cell_fall(scalar) {
            values( " 2.57 ");
        }
        fall_transition(scalar) {
            values( " 0.017 ");
        }
    related_pin : "clk2";
}
```

```
    }
}
} /* end cell */
```

Using Blocks With Interface Timing in Synthesis

To use a cell with interface timing, follow the same procedure used to instantiate technology-specific library cells in a design. See the Design Compiler reference manuals for the details of this procedure. This is a brief review of those steps:

1. Instantiate the block in the design in the same way technology-specific cells are instantiated. Note that cells with interface timing cannot be inferred, because they are black boxes in terms of functionality.
2. For the design to link successfully, add the library containing the block to the `link_library dc_shell` variable.
3. Compile the design.
4. Use the `report_timing` and `report_constraint dc_shell` commands to obtain timing and constraint reports.

You can use multiple libraries in the `link_library dc_shell` variable. Different libraries can use different delay models to describe the timing of their member cells. For example, one library might describe the interface timing of complex blocks that use the nonlinear delay model, whereas another library in the list might use the CMOS delay model.

You can also use multiple libraries in the `target_library dc_shell` variable. The library that contains blocks with interface timing does not need to be included in the `target_library dc_shell` variable unless the library also contains regular ASIC cells that are not black boxes in terms of functionality.

To simulate a design containing a complex sequential block with interface timing, use the functional simulation model provided for the block.

Interpreting Timing Relationships

Synthesis tools interpret each of the timing relationships in the following manner:

Setup or hold arcs

A setup or hold arc between an input pin (clock or data) and a clock edge (rising or falling) implies the existence of an edge-triggered flip-flop with the setup or hold value.

If the setup and hold arcs are specified with respect to the same edge of the clock, one edge-triggered device is implied. Otherwise, two devices are implied, one for each

relationship. In this case, the value of the other constraint for each device is assumed to be 0.0. For example, the value of the hold constraint for the device that checks the setup constraint is 0.0.

Note that a setup or hold relationship of an input pin with respect to a different clock input implies the existence of a separate edge-triggered device.

Output delay and clock edge

An output delay specified with respect to a clock edge implies an edge-triggered device clocked by that clock edge. An output delay arc specified with respect to a different clock pin implies the existence of a separate device (a multistage storage device).

Output delay and clock edge

An output delay specified with respect to a clock edge implies an edge-triggered device clocked by that clock edge. An output delay arc specified with respect to a different clock pin implies the existence of a separate device (a multistage storage device).

Output delay and nonclock input

An output delay with respect to a nonclock input implies the existence of a *direct* combinational path from the input to the output. This arc can coexist with other arcs from clock pins to the same output. Transparent devices and, therefore, time borrowing are not supported.

Other sequential arcs

Synthesis tools do not observe other types of sequential arcs (such as clear, preset, recovery, and removal).

Examples Using Interface Timing Specification

The following example shows how to specify the interface timing of a typical RAM, using the interface timing specification. [Figure 20-8](#) and [Figure 20-9](#) show the timing relationships of the RAM read and write cycles.

Figure 20-8 Read Cycle Timing of the RAM (WR = 1)

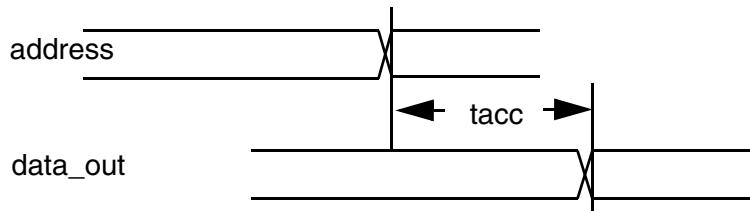
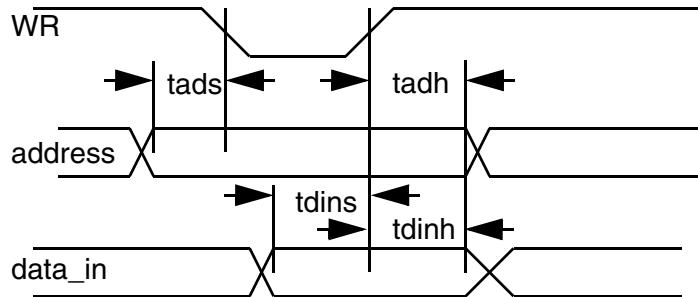


Figure 20-9 Write Cycle Timing of the RAM (WR = 0)



[Example 20-7](#) describes the Library Compiler model of the RAM. Note that the delay between the output data and the address in the read cycle is modeled as a combinational delay. Also note that the WR signal is labeled as a clock.

Example 20-7 Library Compiler Model of RAM in [Figure 20-8](#) and [Figure 20-9](#)

```
cell (RAM_10_8) {
    area : 2300.000000;
    interface_timing : true;
    /* Apply interface timing semantics */
    bus (address) {
        bus_type : "bus10"; /* defined in the library */
        direction : input;
        capacitance : 1.46;
        fanout_load : 1.46;
        timing () /* tads */
            timing_type : setup_falling;
            cell_rise(scalar) {
                values( " 3.20 ");
            }
            cell_fall(scalar) {
                values( " 3.20 ");
            }
            related_pin : "WR";
        }
        timing () /* tadh */
            timing_type : hold_rising;
            cell_rise(scalar) {
                values( " 1.85 ");
            }
            cell_fall(scalar) {
                values( " 1.85 ");
            }
            related_pin : "WR";
    }
}
```

```
bus (data_in) {
    bus_type : "bus8";
    direction : input;
    capacitance : 1.46;
    fanout_load : 1.46;
    timing () /* tdis */
        timing_type : setup_rising;
    cell_rise(scalar) {
        values( " 1.50 ");
    }
    cell_fall(scalar) {
        values( " 1.50 ");
    }
    related_pin : "WR";
}
timing () /* tdih */
    timing_type : hold_rising;
    cell_rise(scalar) {
        values( " 0.65 ");
    }
    cell_fall(scalar) {
        values( " 0.65 ");
    }
    related_pin : "WR";
}
pin (WR) {
    direction : input;
    capacitance : 1.13;
    fanout_load : 1.13;
    clock : true; /* WR pulse is modeled as a clock */
}

bus(data_out){
    bus_type : "bus8";
    direction : output;
    timing () /* tacc*/
        timing_sense : non_unate; /* combinational delay */
    cell_rise(scalar) {
        values( " 5.25 ");
    }
    rise_transition(scalar) {
        values( " 0.020 ");
    }
    cell_fall(scalar) {
        values( " 5.50 ");
    }
    fall_transition(scalar) {
        values( " 0.017 ");
    }
    related_bus_pins : "address";
}
}
```

```
    } /* end cell */
```

Example 20-8 shows a memory model in the interface timing specification format that contains modes.

Example 20-8 Memory Model Showing the Use of Modes.

```
library("ITS_LIB") {
  technology("cmos");
  delay_model : "table_lookup";
  time_unit : "1ps";

  lu_table_template("slewload") {
    variable_1 : "input_net_transition";
    index_1("1.0, 2.0, 3.0");
    variable_2 : "total_output_net_capacitance";
    index_2("1.0, 2.0, 3.0");
  }

  lu_table_template("slewcons") {
    variable_1 : "constrained_pin_transition";
    index_1("1.0, 2.0, 3.0");
    variable_2 : "related_pin_transition";
    index_2("1.0, 2.0, 3.0");
  }
  nom_voltage : 3.0;
  nom_temperature : 100.0;
  type("bus_type1") {
    base_type : array;
    data_type : bit;
    bit_from : 1;
    bit_to : 0;
  }

  cell(ram_core) {
    interface_timing : true;
    area : 100.0;
    mode_definition("ram_modes") {
      mode_value(read1) {
        sdf_cond : "( RWN1 == 1 && CSN1 == 0 )";
        when : "(RWN1 & CSN1')";
      }
    }
  }

  bus("DI1") {
    bus_type : bus_type1;
    direction : input;
    timing() {
      timing_type : "setup_rising";
      sdf_cond : "CSN1 == 0";
      when : "CSN1'";
      rise_constraint("slewcons") {...}
      fall_constraint("slewcons") {...}
    }
  }
}
```

```

        related_pin : "RWN1";
    }
    timing() {
        timing_type : "setup_rising";
        sdf_cond : "RWN1 == 0";
        when : "RWN1'";
        rise_constraint("slewcons") {...}
        fall_constraint("slewcons") {...}
        related_pin : "CSN1";
    }

    pin("DI1[1] DI1[0]") {
        direction : input;
        capacitance : 2.0;
        timing() {
            timing_type : "hold_rising";
            sdf_cond : "CSN1 == 0";
            when : "CSN1'";
            related_pin : "RWN1";

            rise_constraint("slewcons") {...}
            fall_constraint("slewcons") {...}
        }
    }

    timing() {
        timing_type : "hold_rising";
        sdf_cond : "RWN1 == 0";
        when : "RWN1'";
        related_pin : "CSN1";

        rise_constraint("slewcons") {...}
        fall_constraint("slewcons") {...}
    }
}

bus("A1") {
    bus_type : bus_type1;
    direction : input;
    timing() {
        .....
    }
}
bus("DO1") {
    bus_type : bus_type1;
    direction : output;
    timing() {
        timing_sense : "positive_unate";
        mode(ram_modes, "read1");
        rise_transition("slewload") {...}
        fall_transition("slewload") {...}
        cell_rise("slewload") {...}
        cell_fall("slewload") {...}
        related_bus_pins : "A1";
    }
}

```

```
        }
    timing() {
        timing_type : "falling_edge";
        sdf_cond : "RWN1 == 0";
        when : "RWN1'";
        rise_transition("slewload") {...}
        fall_transition("slewload") {...}
        cell_rise("slewload") {...}
        cell_fall("slewload") {...}
        related_pin : "CSN1";
    }
}

bus("DI2") {
    bus_type : bus_type1;
    direction : input;
    timing() {
        ...
    }
}

bus("A2") {
    timing() {
        ...
    }
}
bus("DO2") {
    bus_type : bus_type11;
    direction : output;

    timing() {
        ...
    }
}

pin("RWN1") {
    direction : input;
    capacitance : 5.0;
    clock : true;
    min_pulse_width_low : 1.0;

    timing() {
        ...
    }
}

pin("CSN1") {
    direction : input;
    capacitance : 1.0;
    clock : true;
    min_pulse_width_low : 1.0;

    timing() {
```

```
        ...
    }
}

pin("RWN2") {
    direction : input;
    capacitance : 5.0;
    clock : true;
    min_pulse_width_low : 1.0;
    timing() {
        ...
    }
}
pin("CSN2") {
    direction : input;
    capacitance : 1.0;
    clock : true;
    min_pulse_width_low : 2.0;
    timing() {
        ...
    }
}
}
```


21

Defining I/O Pads

To define I/O pads, you use the `library`, `cell`, and `pin` group attributes that describe input, output, and bidirectional pad cells. The pad modeling features support the Design Compiler synthesis of I/O pads in standard CMOS technologies.

To model I/O pads, you must understand the following concepts covered in this chapter:

- [Special Characteristics of I/O Pads](#)
- [Identifying Pad Cells](#)
- [Describing Multicell Pads](#)
- [Defining Units for Pad Cells](#)
- [Describing Input Pads](#)
- [Describing Output Pads](#)
- [Modeling Wire Load for Pads](#)
- [Internal Isolation Support in I/O Pad Cell Models](#)
- [Programmable Driver Type Support in I/O Pad Cell Models](#)
- [Reporting Pad Information](#)
- [Pad Cell Examples](#)

Special Characteristics of I/O Pads

I/O pads are the special cells at the chip boundaries that allow communication with the world outside the chip. Their characteristics distinguish them from the other cells that make up the core of an integrated circuit. These characteristics must be described in Synopsys technology libraries so that the Design Compiler tool can understand the pads well enough to insert them automatically during synthesis.

Pads typically have longer delays and higher drive capabilities than the cells in an integrated circuit's core. Because of their higher drive, CMOS pads sometimes exhibit noise problems. Slew-rate control is available on output pads to help alleviate this problem.

Pads are fixed resources—a limited number are available on each die size. In addition, special types, such as clock pads, might be more limited than others. These limits must be modeled.

A distinguishing feature of pad cells is the voltage level at which input pads transfer logic 0 or logic 1 signals to the core or at which output pad drivers communicate logic values from the core.

Integrated circuits that communicate with one another must have compatible voltage levels at their pads. Because pads communicate with the world outside the integrated circuit, you must describe the pertinent units of peripheral library properties, such as external load, drive capability, delay, current, power, and resistance. This description makes it easier to design chips from multiple technologies.

Some libraries create logical pads out of multiple cells; such logical pads must be modeled so that they are just as easy to insert automatically on a design as a single-cell pad.

You must capture all these properties in the library to make it possible for the integrated circuit designer to insert the correct pads during synthesis.

Identifying Pad Cells

Use the attributes described in the following sections to specify I/O pads and pad pin behaviors.

Regular Pad Cells

Use the `pad_cell` attribute to tag a cell as an I/O pad. The Design Compiler tool filters pad cells out of the normal core optimization and treats them differently during technology translation.

Example

```
pad_cell : true;
```

Clock Pads

Certain pad and auxiliary pad cells are special and require special treatment. The most important of these is the clock driver pad cell. It is important to identify clock drivers, because the Design Compiler and DFT Compiler tools treat them differently.

To designate a clock pad on a cell tagged with a `pad_cell` or an `auxiliary_pad_cell` attribute, use the `pad_type` attribute statement.

Example

```
pad_type : clock;
```

Pad Pin Attributes

Each cell identified as an I/O pad must have a pin that represents the pad. Additionally, the pin must have a direction indicating whether the cell is an input, output, or bidirectional pad.

Use the `is_pad` attribute on pins or parameterized pins to identify which pins are to be considered logical I/O pad pins. Use the `is_pad` attribute on at least one pin of a cell with a `pad_cell` attribute.

The `direction` attribute identifies whether the pad is an input, output, or bidirectional pad.

Example

```
pin(PAD) {
    direction : output;
    is_pad : true;
...
}
```

If a pin is bidirectional, you can supply multiple `driver_type` attributes for the same pin to model both the output behavior and the input behavior. An example is

```
driver_type : "open_drain pull_up" ;
```

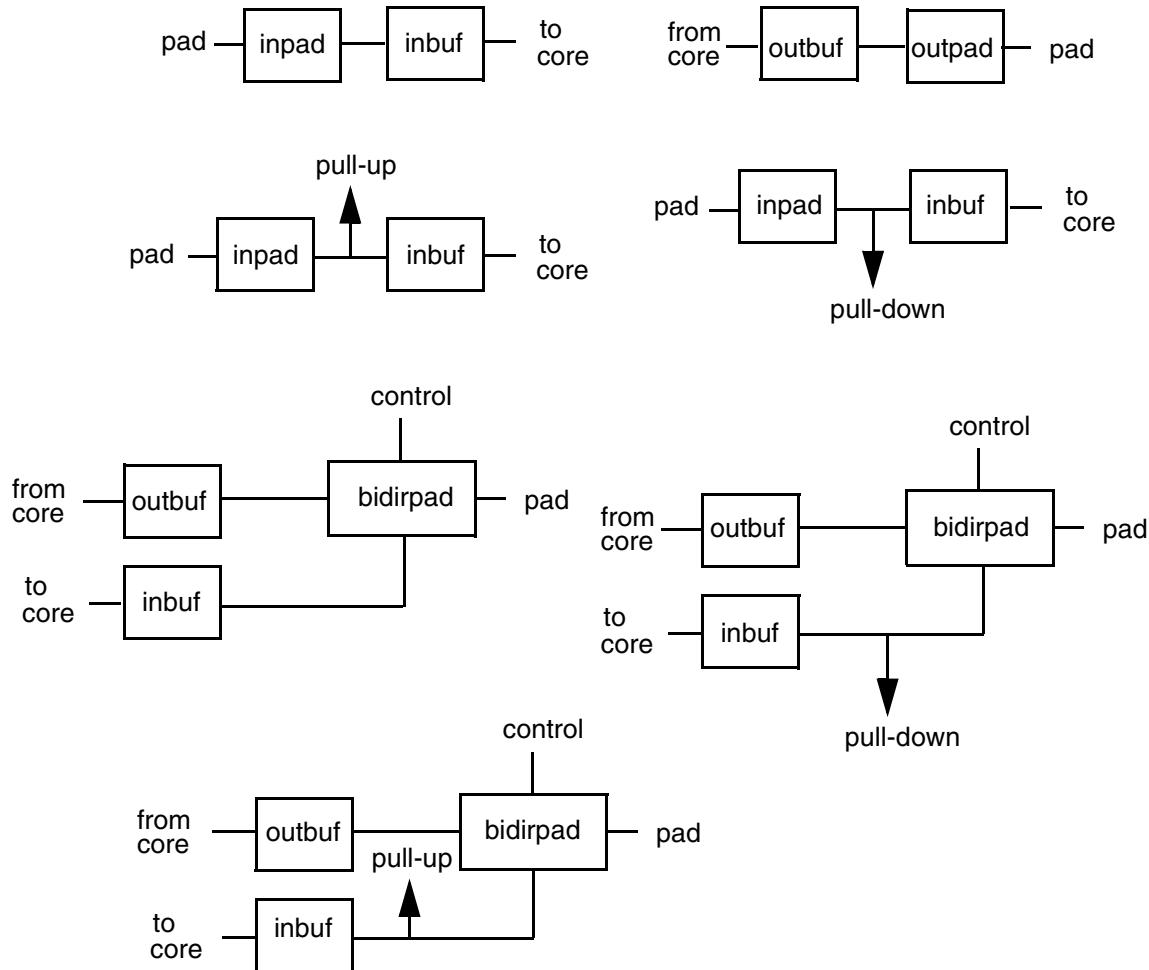
Describing Multicell Pads

I/O pads can be represented either as one cell with many attributes or as a collection of cells, each with attributes affecting the operation of the logical pad. CMOS technologies implement pads with multiple cells—a pad cell and a driver cell, for example.

The first type of library contains a different cell for each possible combination of pad characteristics. The second type contains a smaller number of components, but each type of pad must be individually constructed from these components.

[Figure 21-1](#) shows the different components that make up multicell pads in the second type of library. See [Example 21-9 on page 21-23](#) for the Library Compiler descriptions of these components.

Figure 21-1 Multicell Pads



Auxiliary Pad Cells

If you implement your pads with more than one cell, you can identify those cells that can be used to build up a logical pad with an `auxiliary_pad_cell` attribute. This attribute indicates that the cell can be used as part of a logical pad.

Example

```
auxiliary_pad_cell : true;
```

Identifying such cells helps the Design Compiler tool create I/O pads consisting of more than one cell. Cells that have the `auxiliary_pad_cell` attribute can also be used within the core.

Auxiliary pad cells are used only with pull-up and pull-down devices.

Connecting Pins

If your library uses multicell pads, the Design Compiler tool needs to know which pins on the various cells to connect. This information lets the tool implement the pad properly. Two attributes give this information:

```
multicell_pad_pin
```

Identifies which pins to connect to create a working multicell pad. Use this attribute to flag all the pins to connect on a pad or an auxiliary pad cell.

```
connection_class
```

Indicates which pins to connect to pins on other cells. For example, pull-up cells are generally connected to the network between the input pad cell and the input driver cell. This attribute tells the Design Compiler tool which classes of pins to connect.

Example

```
multicell_pad_pin : true;
connection_class : "INPAD_NETWORK";
```

Defining Units for Pad Cells

To process pads for full-chip synthesis, specify the units of time, capacitance, resistance, voltage, current, and power. The Library Compiler tool uses the following attributes to provide the required unit information:

- `time_unit`
- `capacitive_load_unit`
- `pulling_resistance_unit`
- `voltage_unit`
- `current_unit`
- `leakage_power_unit`

All these attributes are defined at the library level, as described in “[Input-Capacitance Characterization Attributes](#)” on page B-14. These values are required. If you do not supply these attributes, the Design Compiler tool prints a warning.

Units of Time

The downstream tools use the `time_unit` attribute to identify the time unit used in the library.

Example

```
time_unit : 10ps ;
```

Capacitance

The `capacitive_load_unit` attribute defines the capacitance associated with a standard load. If you already represent capacitance values in terms of picofarads or femtofarads, use this attribute to define your base unit. If you represent capacitance in terms of the standard load of an inverter, define the exact capacitance for that inverter—for example, 0.101 pF.

Example

```
capacitive_load_unit( 0.1,ff ) ;
```

Resistance

You must supply a `pulling_resistance` attribute for pull-up and pull-down devices on pads and identify the unit to use with the `pulling_resistance_unit` attribute.

Example

```
pulling_resistance_unit : "1kohm";
```

Voltage

You can use the `input_voltage` and `output_voltage` groups to define a set of input or output voltage ranges for your pads. To define the units of voltage you use for these groups, use the `voltage_unit` attribute. All the attributes defined inside `input_voltage` and `output_voltage` groups are scaled by the value defined for `voltage_unit`. In addition, the `voltage` attribute in the `operating_conditions` groups also represents its values in these units.

Example

```
voltage_unit : "1V";
```

Current

You can define the drive current that can be generated by an output pad and also define the pulling current for a pull-up or pull-down transistor under nominal voltage conditions. Define all current values with the library-level `current_unit` attribute.

Example

```
current_unit : "1uA";
```

Power

The `leakage_power_unit` attribute defines the units of the power values in the library. If this attribute is missing, the leakage-power values are expressed without units.

Example

```
leakage_power_unit : "100uW" ;
```

Describing Input Pads

To represent input pads in your technology library, you must describe the input voltage characteristics and indicate whether hysteresis applies.

The input pad properties are described in the next section. Examples at the end of this chapter describe a standard input buffer, an input buffer with hysteresis, and an input clock buffer.

Voltage Levels

You can use the `input_voltage` group at the library level to define a set of input voltage ranges for your pads. You can then assign this set of voltage ranges to the input pin of a pad cell.

For example, you can define an `input_voltage` group called TTL with a set of high and low thresholds and minimum and maximum voltage levels and use this command in the `pin` group to assign those ranges to the pad cell pin:

```
input_voltage : TTL ;
```

You can include the `vil`, `vih`, `vimin`, and `vimax` attributes in the `input_voltage` group. See “[Input Voltage Group](#)” on page 5-15 for additional information.

Hysteresis

You can indicate an input pad with hysteresis, using the `hysteresis` attribute. The default for this attribute is false. When it is true, the `vil` and `vol` voltage ratings are actual transition points.

Example

```
hysteresis : true;
```

Pads with hysteresis sometimes have derating factors that are different from those of cells in the core. This construct provides derating capabilities and minimum, typical, or maximum timing for cells.

Describing Output Pads

To represent output pads in your technology library, you must describe the output voltage characteristics and the drive-current rating of output and bidirectional pads. Additionally, you must include information about the slew rate of the pad. These output pad properties are described in the sections that follow.

Examples at the end of this chapter show a standard output buffer and a bidirectional pad.

Voltage Levels

You can use the `output_voltage` group at the library level to define a set of output voltage ranges for your pads. You can then use the `output_voltage` attribute at the pin level to assign this set of voltage ranges to the output pin of a pad cell.

Example

```
output_voltage(TTL) {
    vol : 0.4;
    voh : 2.4;
    vomin : -0.3;
    vomax : VDD + 0.3;
}
```

You can include `vol`, `voh`, `vomin`, and `vomax` values in `output_voltage` groups. In this example, an `output_voltage` group called TTL contains a set of high and low thresholds and minimum and maximum voltage levels.

In the `pin` group, you can assign these ranges to the pad cell pin with the `output_voltage` attribute:

```
pin(Z) {
```

```
    direction : output;
    is_pad : true;
    output_voltage : TTL ;
    ...
}
```

See “[output_voltage Group](#)” on page [5-15](#) for additional information.

Drive Current

Output and bidirectional pads in a technology can have different drive-current capabilities. To define the drive current supplied by the pad buffer, use the `drive_current` attribute on an output or bidirectional pad or auxiliary pad pin. The value is in units consistent with the `current_unit` attribute you defined.

Example

```
pin(PAD) {
    direction : output;
    is_pad : true;
    drive_current : 1.0;
}
```

Slew-Rate Control

Slew-rate control limits peak noise and smooths out fast output transitions. The Library Compiler tool implements slew-rate control by using one qualitative attribute—`slew_control`—and the following eight quantitative attributes:

- `rise_current_slope_before_threshold`
- `rise_current_slope_after_threshold`
- `fall_current_slope_before_threshold`
- `fall_current_slope_after_threshold`
- `rise_time_before_threshold`
- `rise_time_after_threshold`
- `fall_time_before_threshold`
- `fall_time_after_threshold`

The `slew_control` attribute accepts one of four possible enumerations: none, low, medium, and high; the default is none. Increasing the slew control level slows down the transition rate. This method is the coarsest way to measure the level of slew-rate control associated with an output pad.

You can define slew-rate control for more detail. The Library Compiler tool accepts the eight quantitative attributes to define a two-piece approximation of the current versus time characteristics (dI/dT) of the pad.

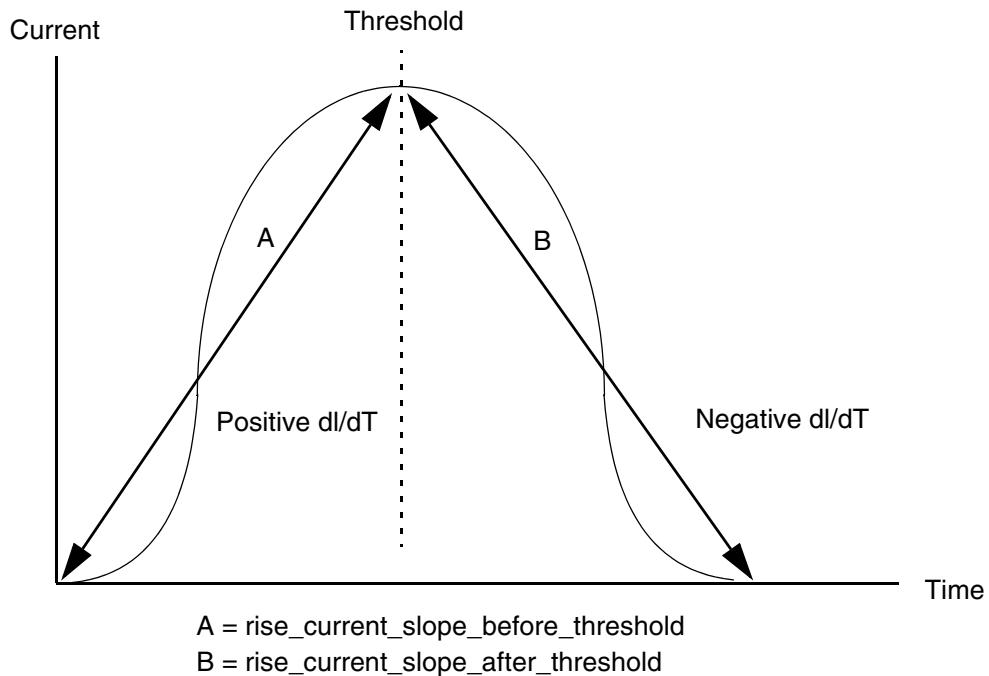
These attributes characterize the dI/dT behavior of an output pad on rising and falling transitions. In addition, two attributes approximate the difference in dI/dT before and after the threshold is reached.

Finally, the attributes define the time intervals for rising and falling transitions from start to threshold (before) and from threshold to finish (after).

For more information about slew-rate control attributes, see “[Slew-Rate Control Attributes](#)” on page 7-56.

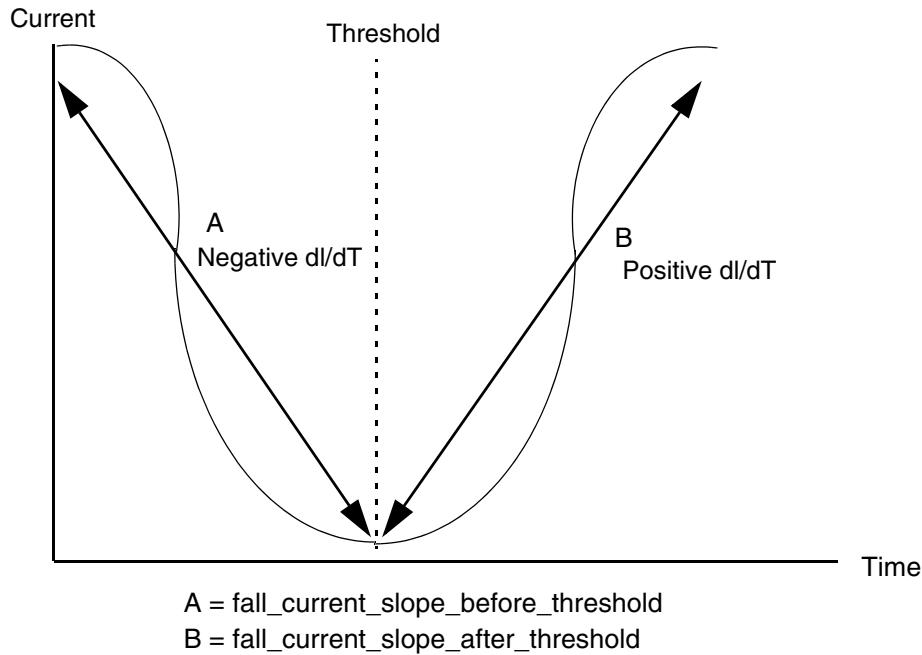
[Figure 21-2](#) depicts the `rise_current_slope` attributes. The A value is a positive number representing a linear approximation of the change of current as a function of time from the beginning of the rising transition to the threshold point. The B value is a negative number representing a linear approximation of the current change over time from the point at which the rising transition reaches the threshold to the end of the transition.

Figure 21-2 Slew-Rate Attributes—Rising Transitions



For falling transitions, the graph is reversed: A is negative and B is positive, as shown in [Figure 21-3](#).

Figure 21-3 Slew-Rate Attributes—Falling Transitions



[Example 21-1](#) shows the slew-rate control attributes:

Example 21-1 Slew-Rate Control Attributes

```
pin(PAD) {
    is_pad : true;
    direction : output;
    output_voltage : GENERAL;
    slew_control : high;
    rise_current_slope_before_threshold : 0.18;
    rise_time_before_threshold : 0.8;
    rise_current_slope_after_threshold : -0.09;
    rise_time_after_threshold : 2.4;
    fall_current_slope_before_threshold : -0.14;
    fall_time_before_threshold : 0.55;
    fall_current_slope_after_threshold : 0.07;
    fall_time_after_threshold : 1.8;
    ...
}
```

Modeling Wire Load for Pads

You can define several `wire_load` groups that contain all the information the Design Compiler tool needs to estimate interconnect wiring delays. Estimated wire loads for pads can be significantly different from those of core cells.

The wire load model for the net connecting an I/O pad to the core needs to be handled separately, because such a net is usually longer than most other nets in the circuit. Some pad nets extend completely across the chip.

You can define the `wire_load` group, which you want to use for wire load estimation, on the pad ring. For example, name the group `Pad_WireLoad`. Add a level of hierarchy by placing the pads in the top level and by placing all the core circuitry at a lower level. Then, during the Design Compiler session, you can define the `Pad_WireLoad` model when you compile the hierarchical level containing the pads:

```
dc_shell> set_wire_load Pad_WireLoad
```

A different model is defined for the core hierarchical level.

Internal Isolation Support in I/O Pad Cell Models

Apply the `is_isolated` attribute to a pin of an I/O pad cell, to indicate that the pin is internally isolated. Use the optional `isolation_enable_condition` attribute to specify the Boolean condition of isolation for the internally isolated input and inout pins of the pad cell.

For more information about I/O pad cell internal isolation modeling, see “[I/O Pad Cell Internal Isolation Modeling](#)” on page 13-138.

Programmable Driver Type Support in I/O Pad Cell Models

To support pull-up and pull-down circuit structures, the Liberty models for I/O pad cells support pull-up and pull-down driver information using the `driver_type` attribute with the `pull_up` or `pull_down` values.

Liberty syntax also supports conditional (programmable) pull-up and pull-down driver information for I/O pad cells. The programmable pin syntax has also been extended to other `driver_type` attribute values, such as `bus_hold`, `open_drain`, `open_source`, `resistive`, `resistive_0`, and `resistive_1`.

Syntax

The following syntax supports programmable driver types in I/O pad cell models. Unlike the nonprogrammable driver type support, the programmable driver type support allows you to specify more than one driver type within a pin.

```
pin (pin_name) { /* programmable driver type pin */
    ...
    pull_up_function : "function string";
    pull_down_function : "function string";
```

```

bus_hold_function : "function string";
open_drain_function : "function string";
open_source_function : "function string";
resistive_function : "function string";
resistive_0_function : "function string";
resistive_1_function : "function string";
...
}

```

Programmable Driver Type Functions

The functions in [Table 21-1](#) have been introduced on top of (as an extension of) the existing `driver_type` attribute to support programmable pins. These driver type functions help model the programmable driver types. The same rules that apply to nonprogrammable driver types also apply to these functions.

Table 21-1 Programmable Driver Type Functions

Programmable driver type	Applicable on pin types
<code>pull_up_function</code>	Input, output and inout
<code>pull_down_function</code>	Input, output and inout
<code>bus_hold_function</code>	Inout
<code>open_drain_function</code>	Output and inout
<code>open_source_function</code>	Output and inout
<code>resistive_function</code>	Output and inout
<code>resistive_0_function</code>	Output and inout
<code>resistive_1_function</code>	Output and inout

With the exception of `pull_up_function` and `pull_down_function`, if any of the driver type functions in [Table 21-1](#) is specified on an inout pin, it is used only for output pins.

The following rules apply to programmable driver type functions (as well as nonprogrammable driver types in I/O pad cell models):

- The attribute can be applied to pad cell only.
- Only the input and inout pin can be specified in the function string.
- The function string is a Boolean function of input pins.

The following rules apply to an inout pin:

- If `pull_up_function` or `pull_down_function` and `open_drain_function` are specified within the same inout pin, `pull_up_function` or `pull_down_function` is used for the input pins.
- If `bus_hold_function` is specified on an inout pin, it is used for input and output pins.

Example

[Example 21-2](#) models a programmable driver type in an I/O pad cell.

Example 21-2 Example of Programmable Driver Type

```
library(cond_pull_updown_example) {
delay_model : table_lookup;

time_unit : 1ns;
voltage_unit : 1V;
capacitive_load_unit (1.0, pf);
current_unit : 1mA;

cell(conditional_PU_PD) {
    dont_touch : true ;
    dont_use : true ;
    pad_cell : true ;
    pin(IO) {
        drive_current      : 1 ;
        min_capacitance   : 0.001 ;
        min_transition    : 0.0008 ;
        is_pad            : true ;
        direction         : inout ;
        max_capacitance   : 30 ;
        max_fanout        : 2644 ;
        function          : "(A*ETM')+(TA*ETM)" ;
        three_state        : "(TEN*ETM')+(EN*ETM)" ;
        pull_up_function   : "(!P1 * !P2)" ;
        pull_down_function : "( P1 * P2)" ;
        capacitance       : 2.06649 ;
        timing() {
            related_pin : "IO A ETM TEN TA" ;
            cell_rise(scalar) {
                values("0" ) ;
            }
            rise_transition(scalar) {
                values("0" ) ;
            }
            cell_fall(scalar) {
                values("0" ) ;
            }
            fall_transition(scalar) {
                values("0" ) ;
            }
        }
    }
}
```

```
        }
    }
timing() {

    timing_type : three_state_disable;
    related_pin : "EN ETM TEN" ;
    cell_rise(scalar) {
        values("0" ) ;
    }
    rise_transition(scalar) {
        values("0" ) ;
    }
    cell_fall(scalar) {
        values("0" ) ;
    }
    fall_transition(scalar) {
        values("0" ) ;
    }
}

pin(ZI) {
    direction : output;
    function      : "IO" ;
    timing() {
        related_pin : "IO" ;
        cell_rise(scalar) {
            values("0" ) ;
        }
        rise_transition(scalar) {
            values("0" ) ;
        }
        cell_fall(scalar) {
            values("0" ) ;
        }
        fall_transition(scalar) {
            values("0" ) ;
        }
    }
}
pin(A) {
    direction : input;
    capacitance : 1.0;
}
pin(EN) {
    direction : input;
    capacitance : 1.0;
}
pin(TA) {
    direction : input;
    capacitance : 1.0;
}
pin(TEN) {
    direction : input;
```

```

        capacitance : 1.0;
    }
    pin(ETM) {
        direction : input;
        capacitance : 1.0;
    }
    pin(P1) {
        direction : input;
        capacitance : 1.0;
    }
    pin(P2) {
        direction : input;
        capacitance : 1.0;
    }
} /* End cell conditional_PU_PD */
} /* End Library */

```

Reporting Pad Information

The `report_lib` command prints information about the `voltage` groups and units defined in your library, as shown in [Example 21-3](#). You can print a report for the library as a whole or for only the cells you select. For more information about the `report_lib` command, see the man page.

Example 21-3 Example Library Report

```
*****
Report : library
Library: testpad
Version: 2000.05
Date   : Wed August 11 16:10:03 2005
*****

Library Type          : Technology
Tool Created          : 2005.05
Date Created          : Wed August 11 20:05:36 2005
Library Version       : 2.110100
Time Unit             : 1ns
Capacitive Load Unit : 0.100000ff
Pulling Resistance Unit: 1kilo-ohm
Voltage Unit          : 1V
Current Unit          : 1uA
Bus Naming Style      : %s[%d] (default)
...
Input Voltages:
```

Name	Vil	Vih	Vimin	Vimax
TTL	0.80	2.00	-0.30	VDD + 0.300
TTL_SCHMITT	0.80	2.00	-0.30	VDD + 0.300
CMOS	1.50	3.50	-0.30	VDD + 0.300
CMOS_SCHMITT	1.00	4.00	-0.30	VDD + 0.300

Output Voltages:

Name	Vol	Voh	Vomin	Vomax
GENERAL	0.40	2.40	-0.30	VDD + 0.300

Wire Loading Model:

No wire loading specified.

Wire Loading Model Mode: top.

Pad Cell Examples

These are examples of input, clock, output, and bidirectional pad cells.

Input Pads

The input pad definition in [Example 21-4](#) represents a standard input buffer, and [Example 21-5](#) lists an input buffer with hysteresis. [Example 21-6](#) shows an input clock buffer.

Example 21-4 Input Buffer

```
library (example1) {
    date : "August 14, 2015" ;
    revision : 2015.05;
    ...
    time_unit : "1ns";
    voltage_unit : "1V";
    current_unit : "1uA";
    pulling_resistance_unit : "1kohm";
    capacitive_load_unit( 0.1,ff );
    ...
    define_cell_area(bond_pads,pad_slots);
    define_cell_area(driver_sites,pad_driver_sites);
    ...
    input_voltage(CMOS) {
        vil : 1.5;
        vih : 3.5;
        vimin : -0.3;
        vimax : VDD + 0.3;
    }
    ...
    /***** INPUT PAD*****/
    cell(INBUF) {
        area : 0.000000;
        pad_cell : true;
```

```

bond_pads : 1;
driver_sites : 1;
pin(PAD) {
    direction : input;
    is_pad : true;
    input_voltage : CMOS;
    capacitance : 2.500000;
    fanout_load : 0.000000;
}
pin(Y) {
    direction : output;
    function : "PAD";
    timing() {
        cell_rise(scalar) {
            values( " 3.07 ");
        }
        rise_transition(scalar) {
            values( " 0.50 ");
        }
        cell_fall(scalar) {
            values( " 2.95 ");
        }
        fall_transition(scalar) {
            values( " 0.50 ");
        }
        related_pin :"PAD";
    }
}

```

Example 21-5 Input Buffer With Hysteresis

```

library (example1) {
    date : "August 14, 2015" ;
    revision : 2015.05;
    ...
    time_unit : "1ns";
    voltage_unit : "1V";
    current_unit : "1uA";
    pulling_resistance_unit : "1kohm";
    capacitive_load_unit( 0.1,ff );
    ...
    input_voltage(CMOS_SCHMITT) {
        vil : 1.0;
        vih : 4.0;
        vimin : -0.3;
        vimax : VDD + 0.3;
    }
    ...
    /*INPUT PAD WITH HYSTERESIS*/
    cell(INBUFH) {
        area : 0.000000;
        pad cell : true;

```

```

pin(PAD ) {
    direction : input;
    is_pad : true;
    hysteresis : true;
    input_voltage : CMOS_SCHMITT;
    capacitance : 2.500000;
    fanout_load : 0.000000;
}
pin(Y ) {
    direction : output;
    function : "PAD";
    timing() {
        cell_rise(scalar) {
            values( " 3.07 ");
        }
        rise_transition(scalar) {
            values( " 0.50 ");
        }
        cell_fall(scalar) {
            values( " 2.95 ");
        }
        fall_transition(scalar) {
            values( " 0.50 ");
        }
        related_pin :"PAD";
    }
}
}
}

```

Example 21-6 Input Clock Buffer

```

library (example1) {
    date : "August 12, 2015" ;
    revision : 2015.05;
    ...
    time_unit : "1ns";
    voltage_unit : "1V";
    current_unit : "1uA";
    pulling_resistance_unit : "1kohm";
    capacitive_load_unit( 0.1,ff );
    ...
    input_voltage(CMOS) {
        vil : 1.5;
        vih : 3.5;
        vimin : -0.3;
        vimax : VDD + 0.3;
    }
    ...
    **** CLOCK INPUT BUFFER ****
    cell(CLKBUF) {
        area : 0.000000;
        pad_cell : true;
    }
}

```

```

pad_type : clock;
pin(PAD ) {
    direction : input;
    is_pad : true;
    input_voltage : CMOS;
    capacitance : 2.500000;
    fanout_load : 0.000000;
}
pin(Y ) {
    direction : output;
    function : "PAD";
    max_fanout : 2000.000000;
    timing() {
        cell_rise(scalar) {
            values( " 5.70 ");
        }
        rise_transition(scalar) {
            values( " 0.009921 ");
        }
        cell_fall(scalar) {
            values( " 6.90 ");
        }
        fall_transition(scalar) {
            values( " 0.010238 ");
        }
        related_pin :"PAD";
    }
}
}
}

```

Output Pads

The output pad definition in [Example 21-7](#) represents a standard output buffer.

Example 21-7 Output Buffer

```

library (example1) {
    date : "August 12, 2015" ;
    revision : 2015.05;
    ...
    time_unit : "1ns";
    voltage_unit : "1V";
    current_unit : "1uA";
    pulling_resistance_unit : "1kohm";
    capacitive_load_unit( 0.1,ff );
    ...
    output_voltage(GENERAL) {
        vol : 0.4;
        voh : 2.4;
        vomin : -0.3;
        vomax : VDD + 0.3;
    }
}

```

```
***** OUTPUT PAD *****/
cell(OUTBUF) {
    area : 0.000000;
    pad_cell : true;
    pin(D) {
        direction : input;
        capacitance : 1.800000;
    }
    pin(PAD) {
        direction : output;
        is_pad : true;
        drive_current : 2.0;
        output_voltage : GENERAL;
        function : "D";
        timing() {
            cell_rise(scalar) {
                values( " 8.487 ");
            }
            rise_transition(scalar) {
                values( " 0.16974 ");
            }
            cell_fall(scalar) {
                values( " 9.347 ");
            }
            fall_transition(scalar) {
                values( " 0.18696 ");
            }
            related_pin :"D";
        }
    }
}
```

Bidirectional Pad

[Example 21-8](#) shows a bidirectional pad cell with three-state enable.

Example 21-8 Bidirectional Pad

```
library (example1) {
    date : "August 12, 2015" ;
    revision : 2015.08;
    ...
    time_unit : "1ns";
    voltage_unit : "1V";
    current_unit : "1uA";
    pulling_resistance_unit : "1kohm";
    capacitive_load_unit( 0.1,ff );
    ...
    output_voltage(GENERAL) {
        vol : 0.4;
        voh : 2.4;
        vomin : -0.3;
```

```
        vomax : VDD + 0.3;
    }
/** BIDIRECTIONAL PAD *****/
cell(BIBUF) {
    area : 0.000000;
    pad_cell : true;
    pin(E D ) {
        direction : input;
        capacitance : 1.800000;
    }
    pin(Y ) {
        direction : output;
        function : "PAD";
        driver_type : "open_source pull_up";
        pulling_resistance : 10000;
        timing() {
            cell_rise(scalar) {
                values( " 3.07 ");
            }
            rise_transition(scalar) {
                values( " 0.50 ");
            }
            cell_fall(scalar) {
                values( " 2.95 ");
            }
            fall_transition(scalar) {
                values( " 0.50 ");
            }
            related_pin :"PAD";
        }
    }
    pin(PAD ) {
        direction : inout;
        is_pad : true;
        drive_current : 2.0;
        output_voltage : GENERAL;
        input_voltage : CMOS;
        function : "D";
        three_state : "E";
        timing() {
            cell_rise(scalar) {
                values( " 8.483 ");
            }
            rise_transition(scalar) {
                values( " 0.34686 ");
            }
            cell_fall(scalar) {
                values( " 19.065 ");
            }
            fall_transition(scalar) {
                values( " 0.3813 ");
            }
            related_pin :"E";
        }
    }
}
```

```

        }
        timing() {
            cell_rise(scalar) {
                values( " 17.466 ");
            }
            rise_transition(scalar) {
                values( " 0.16974 ");
            }
            cell_fall(scalar) {
                values( " 9.348 ");
            }
            fall_transition(scalar) {
                values( " 0.18696 ");
            }
            related_pin :"D";
        }
    }
}

```

Components for Multicell Pads

[Example 21-9](#) shows a library that allows multicell pad configurations.

Example 21-9 Multicell Pad Components

```

library (multicell_pad_example) {
    time_unit : "1ns";
    voltage_unit : "1V";
    current_unit : "1mA";
    pulling_resistance_unit : "1kohm";
    default_connection_class : "some_default";
    cell(INPAD) {
        area : 1;
        pad_cell : true;
        pin( PAD ) {
            direction : input;
            capacitance : 1.0;
            is_pad : true ;
        }
        pin( ToInBuf ) {
            direction : output;
            multicell_pad_pin : true;
            connection_class : "INPAD_NETWORK";
            timing() {
        }
        cell(OUTPAD) {
            area : 1;
            pad_cell : true;
            pin( FromOutBuf ) {
                direction : input;
                capacitance : 1.0;
                multicell_pad_pin : true;

```

```
        connection_class : "OUTPAD_NETWORK";
    }
    pin(PAD ) {
        direction : output;
        is_pad : true;
        drive_current : 1.0;
    }
}
cell( BIDIPAD ) {
    area : 1;
    pad_cell : true;
    pin( FromOutBuf ) {
        direction : input;
        capacitance : 1.0;
        multicell_pad_pin : true;
        connection_class : "OUTPAD_NETWORK";
    }
    pin( ToInBuf ) {
        direction : output;
        multicell_pad_pin : true;
        connection_class : "INPAD_NETWORK";
    }
    pin( Control ) {
        direction : input;
        capacitance : 1.0;
    }
    pin( PAD ) {
        direction : inout;
        is_pad : true;
        drive_current : 1.0;
    }
}
cell( INBUF ) {
    area : 1;
    auxiliary_pad_cell : true;
    pin( From_In_Pad ) {
        direction : input;
        capacitance : 1.0;
        multicell_pad_pin : true;
        connection_class : "INPAD_NETWORK";
    }
    pin( ToCore ) {
        direction : output;
    }
}
cell( OUTBUF ) {
    area : 1;
    auxiliary_pad_cell : true;
    pin( To_Out_Pad ) {
        direction : output;
        multicell_pad_pin : true;
        connection_class : "OUTPAD_NETWORK";
    }
}
```

```

        pin( FromCore ) {
            capacitance : 1.0;
            direction : input;
        }
    }
    cell( PULLUP ) {
        area : 1;
        auxiliary_pad_cell : true;
        pin( PULLING_PIN ) {
            direction : output;
            driver_type : pull_up;
            multicell_pad_pin : true;
            connection_class : "INPAD_NETWORK";
            pulling_resistance : 1000; /* 1000 ohms */
        }
    }
    cell( PULLDOWN ) {
        area : 1;
        auxiliary_pad_cell : true;
        pin( PULLING_PIN ) {
            direction : output;
            driver_type : pull_down;
            multicell_pad_pin : true;
            connection_class : "INPAD_NETWORK";
            pulling_resistance : 1000; /* 1000 ohms */
        }
    }
}

```

Cell with contention_condition and x_function

[Example 21-10](#) shows a cell with the `contention_condition` attribute, which specifies contention-causing conditions and the `x_function` attribute, which describes the X behavior of the pin. See “[contention_condition Attribute](#)” on page 7-7 and the “[x_function Attribute](#)” description in “[Describing Clock Pin Functions](#)” on page 7-47 for more information about these attributes.

Note:

DFT Compiler uses only the `contention_condition` attribute. Formality uses only the `x_function` attribute.

Example 21-10 Cell With contention_condition and x_function Attributes

```

default_fanout_load : 0.1;
default inout_pin_cap : 0.1;
default input_pin_cap : 0.1;
default output_pin_cap : 0.1;

capacitive_load_unit(1, pf);
pulling_resistance_unit : 1ohm;

```

```
voltage_unit : 1V;
current_unit : 1mA;
time_unit : 1ps;

cell (cell_a) {
    area : 1;
    contention_condition : "!ap & an";

    pin (ap, an) {
        direction : input;
        capacitance : 1;
    }
    pin (io) {
        direction : output;
        function : "!ap & !an";
        three_state : "ap & !an";
        x_function : "!ap & an";
        timing() {
            related_pin : "ap an";
            timing_type : three_state_disable;
            cell_rise(scalar) {
                values( " 0.10 ");
            }
            rise_transition(scalar) {
                values( " 0.01 ");
            }
            cell_fall(scalar) {
                values( " 0.10 ");
            }
            fall_transition(scalar) {
                values( " 0.01 ");
            }
        }
        timing() {
            related_pin : "ap an";
            timing_type : three_state_enable;
            cell_rise(scalar) {
                values( " 0.10 ");
            }
            rise_transition(scalar) {
                values( " 0.01 ");
            }
            cell_fall(scalar) {
                values( " 0.10 ");
            }
            fall_transition(scalar) {
                values( " 0.01 ");
            }
        }
    }
    pin (z) {
        direction : output;
        function : "!ap & !an";
    }
}
```

```
        x_function : "!ap & an | ap & !an";
    }
}
```


A

Clock-Gating Integrated Cell Circuits

This appendix contains the following information for the `clock_gating_integrated_cell` attribute:

- A list and description of the options available
- Schematics of the circuits represented by the value options available
- An example of each of the value options available

It includes the following sections:

- [List and Description of Options](#)
- [Schematics and Examples](#)

List and Description of Options

[Table A-1](#) lists and describes the `clock_gating_integrated_cell` attribute values.

Specify the `clock_gating_integrated_cell` attribute, as shown:

```
clock_gating_integrated_cell : value ;
```

where `value` is a concatenation of up to four strings that describe the cell's function. The valid values are listed in [Table A-1](#). For more information about the cells described in this table, see [“Schematics and Examples” on page A-7](#).

The Power Compiler tool supports these cells as well as the generic integrated clock-gating cells in [Table A-2 on page A-5](#). After you model a cell using the integrated clock-gating syntax and specify a value, the Library Compiler tool derives one of the structures in [Table A-1](#) automatically.

Table A-1 Values for the `clock_gating_integrated_cell` Attribute

<code>clock_gating_integrated_cell</code> values	Integrated cell must contain
<code>latch_posedge</code>	Latch-based gating logic Logic appropriate for rising-edge-triggered registers
<code>latch_posedge_precontrol</code>	Latch-based gating logic Logic appropriate for rising-edge-triggered registers Test-control logic located before the latch
<code>latch_posedge_postcontrol</code>	Latch-based gating logic Logic appropriate for rising-edge-triggered registers Test-control logic located after the latch
<code>latch_posedge_precontrol_obs</code>	Latch-based gating logic Logic appropriate for rising-edge-triggered registers Test-control logic located before the latch Observability logic
<code>latch_posedge_postcontrol_obs</code>	Latch-based gating logic Logic appropriate for rising-edge-triggered registers Test-control logic located after the latch Observability logic
<code>latch_negedge</code>	Latch-based gating logic Logic appropriate for falling-edge-triggered registers

Table A-1 Values for the clock_gating_integrated_cell Attribute (Continued)

clock_gating_integrated_cell values	Integrated cell must contain
latch_negedge_precontrol	<p>Latch-based gating logic</p> <p>Logic appropriate for falling-edge-triggered registers</p> <p>Test-control logic located before the latch</p>
latch_negedge_postcontrol	<p>Latch-based gating logic</p> <p>Logic appropriate for falling-edge-triggered registers</p> <p>Test-control logic located after the latch</p>
latch_negedge_precontrol_obs	<p>Latch-based gating logic</p> <p>Logic appropriate for falling-edge-triggered registers</p> <p>Test-control logic located before the latch</p> <p>Observability logic</p>
latch_negedge_postcontrol_obs	<p>Latch-based gating logic</p> <p>Logic appropriate for falling-edge-triggered registers</p> <p>Test-control logic located after the latch</p> <p>Observability logic</p>
none_posedge	<p>Latch and flip-flop free gating logic</p> <p>Logic appropriate for rising-edge-triggered registers</p>
none_posedge_control	<p>Latch and flip-flop free gating logic</p> <p>Logic appropriate for rising-edge-triggered registers</p> <p>Test-control logic (no latch and no flip-flop)</p>
none_posedge_control_obs	<p>Latch and flip-flop free gating logic</p> <p>Logic appropriate for rising-edge-triggered registers</p> <p>Test-control logic (no latch and no flip-flop)</p> <p>Observability logic</p>
none_negedge	<p>Latch and flip-flop free gating logic</p> <p>Logic appropriate for falling-edge-triggered registers</p>

Table A-1 Values for the clock_gating_integrated_cell Attribute (Continued)

clock_gating_integrated_cell values	Integrated cell must contain
none_nededge_control	<p>Latch and flip-flop free gating logic</p> <p>Logic appropriate for falling-edge-triggered registers</p> <p>Test-control logic (no latch and no flip-flop)</p>
none_nededge_control_obs	<p>Latch and flip-flop free gating logic</p> <p>Logic appropriate for falling-edge-triggered registers</p> <p>Test-control logic (no latch and no flip-flop)</p> <p>Observability logic</p>

[Table A-2 on page A-5](#) lists additional latch-based structures that are supported by the `clock_gating_integrated_cell` attribute. The structure of the cells in this table is similar to the cells in [Table A-1 on page A-2](#). However, you must use the following generic integrated clock-gating (ICG) syntax to model these cells:

```
clock_gating_integrated_cell : generic ;
```

When you specify the generic value, the Library Compiler tool determines the clock-gating structure based on the `latch` group and the `function` attribute values specified in the cell.

Note:

The generic integrated clock-gating syntax supports only the modeling of cell functional information using the `latch` group and the `function` attribute. If you specify `statetable` or `state_function` to model generic integrated clock-gating cells, the Library Compiler tool generates compilation errors.

The Power Compiler tool supports generic integrated clock-gating cells as well as the cells in [Table A-1](#). After you model a cell using the generic integrated clock-gating syntax, the Library Compiler tool derives one of the structures in [Table A-1](#) or [Table A-2](#) automatically.

To see circuit schematics for the cells listed in [Table A-2](#), see “[Generic Integrated Clock-Gating Schematics](#)” on page [A-39](#). (This chapter does not include .lib examples for these cells.)

Table A-2 Additional Latch-Based Structures Supported by the clock_gating_integrated_cell Attribute

Derived cell	Type of gated register	Gating logic used	Control point position	obs port	Active-low enabled
latch_posedgeactivelow	posedge	enl.clk	none	no	yes
latch_posedgeactivelow_precontrol	posedge	enl.clk	before seq	no	yes
latch_posedgeactivelow_postcontrol	posedge	enl.clk	after seq	no	yes
latch_posedgeactivelow_precontrol_obs	posedge	enl.clk	before seq	yes	yes
latch_posedgeactivelow_postcontrol_obs	posedge	enl.clk	after seq	yes	yes
latch_negedgeactivelow	negedge	!enl + clk	none	no	yes
latch_negedgeactivelow_precontrol	negedge	!enl + clk	before seq	no	yes
latch_negedgeactivelow_postcontrol	negedge	!enl + clk	after seq	no	yes
latch_negedgeactivelow_precontrol_obs	negedge	!enl + clk	before seq	yes	yes
latch_negedgeactivelow_postcontrol_obs	negedge	!enl + clk	after seq	yes	yes
latch_posedge_invgclk	posedge	!(enl.clk)	none	no	no
latch_posedge_precontrol_invgclk	posedge	!(enl.clk)	before seq	no	no
latch_posedge_postcontrol_invgclk	posedge	!(enl.clk)	after seq	no	no

Table A-2 Additional Latch-Based Structures Supported by the `clock_gating_integrated_cell` Attribute (Continued)

Derived cell	Type of gated register	Gating logic used	Control point position	obs port	Active-low enabled
latch_posedge_precontrol_obs_invclk	posedge	!(enl.clk)	before seq	yes	no
latch_posedge_postcontrol_obs_invclk	posedge	!(enl.clk)	after seq	yes	no
latch_negedge_invclk	negedge	enl. !clk	none	no	no
latch_negedge_precontrol_invclk	negedge	enl. !clk	before seq	no	no
latch_negedge_postcontrol_invclk	negedge	enl. !clk	after seq	no	no
latch_negedge_precontrol_obs_invclk	negedge	enl. !clk	before seq	yes	no
latch_negedge_postcontrol_obs_invclk	negedge	enl. !clk	after seq	yes	no
latch_posedgeactivelow_invclk	posedge	!enl + !clk	none	no	yes
latch_posedgeactivelow_precontrol_invclk	posedge	!enl + !clk	before seq	no	yes
latch_posedgeactivelow_postcontrol_invclk	posedge	!enl + !clk	after seq	no	yes
latch_posedgeactivelow_precontrol_obs_invclk	posedge	!enl + !clk	before seq	yes	yes
latch_posedgeactivelow_postcontrol_obs_invclk	posedge	!enl + !clk	after seq	yes	yes
latch_negedgeactivelow_invclk	negedge	enl . !clk	none	no	yes
latch_negedgeactivelow_precontrol_invclk	negedge	enl . !clk	before seq	no	yes

Table A-2 Additional Latch-Based Structures Supported by the `clock_gating_integrated_cell` Attribute (Continued)

Derived cell	Type of gated register	Gating logic used	Control point position	obs port	Active-low enabled
latch_nedgedeactive_low_postcontrol_invclk	nededge	enl . !clk	after seq	no	yes
latch_nedgedeactive_low_precontrol_obs_invclk	nededge	enl . !clk	before seq	yes	yes
latch_nedgedeactive_low_postcontrol_obs_invclk	nededge	enl . !clk	after seq	yes	yes

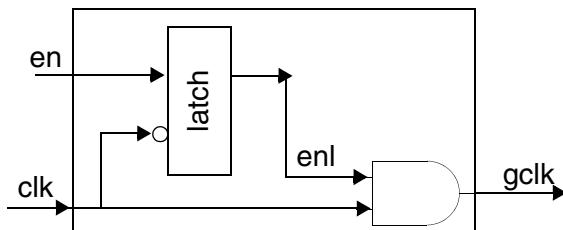
Schematics and Examples

This section shows circuit schematics and examples for the options listed in [Table A-1](#). See “[Generic Integrated Clock-Gating Schematics](#)” on page [A-39](#) to see circuit schematics for the derived integrated clock-gating cell structures listed in [Table A-2](#).

latch_posedge Option

[Figure A-1](#) shows the circuit of the `latch_posedge` option for the `clock_gating_integrated_cell` attribute, and [Example A-1](#) demonstrates the use of this option.

Figure A-1 latch_posedge Circuit



Example A-1 latch_posedge Option

```
cell(CGLP) {
  area : 1;
  clock_gating_integrated_cell : "latch_posedge";
  dont_use : true;
```

```

statetable(" CLK EN ", "ENL ") {
    table : " L  L  : -  : L , \
              L  H  : -  : H , \
              H  -  : -  : N ";
}
pin(ENL) {
    direction : internal;
    internal_node : "ENL";
}
pin(EN) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_enable_pin : true;
    timing() {
        timing_type : setup_rising;
        cell_rise(scalar) {values( " 0.4 ");}
        cell_fall(scalar) {values( " 0.4 ");}
        related_pin : "CLK";
    }
    timing() {
        timing_type : hold_rising;
        cell_rise(scalar) {values( " 0.4 ");}
        cell_fall(scalar) {values( " 0.4 ");}
        related_pin : "CLK";
    }
}
pin(CLK) {
    direction : input;
    capacitance : 0.031419;
    clock_gate_clock_pin : true;
    min_pulse_width_low : 0.319;
}
pin(GCLK) {
    direction : output;
    state_function : "CLK * ENL";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
    timing() {
        timing_sense : positive_unate;
        cell_rise(scalar) {values( " 0.48 ");}
        cell_fall(scalar) {values( " 0.77 ");}
        rise_transition(scalar) {values( " 0.1443 ");}
        fall_transition(scalar) {values( " 0.0523 ");}
        related_pin : "CLK";
    }
    internal_power () {
        rise_power(l14X3) {
            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(l14X3) {
            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", \
                  "0.147, 0.174, 0.304", \
                  "0.177, 0.204, 0.334", \
                  "0.199, 0.219, 0.297");
        }
    }
}

```

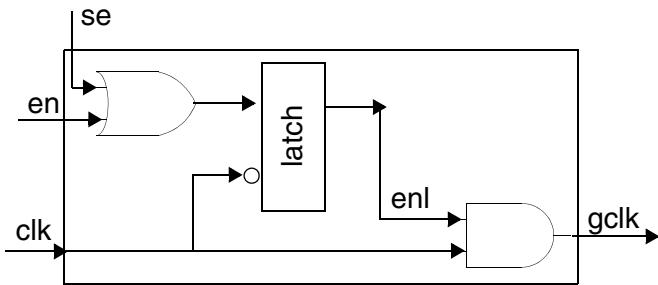
```

        "0.133, 0.151, 0.238", \
        "0.151, 0.186, 0.279", \
        "0.160, 0.190, 0.217");
    }
    related_pin : "CLK EN" ;
}
}
}
```

latch_posedge_precontrol Option

[Figure A-2](#) shows the circuit of the `latch_posedge_precontrol` option for the `clock_gating_integrated_cell` attribute, and [Example A-2](#) demonstrates the use of this option.

Figure A-2 latch_posedge_precontrol Circuit



Example A-2 latch_posedge_precontrol Option

```

cell(CGLPC) {
    area : 1;
    clock_gating_integrated_cell : "latch_posedge_precontrol";
    dont_use : true;
    statetable(" CLK EN SE", "ENL ") {
        table : " L  L  L : - : L , \
                   L  L  H : - : H , \
                   L  H  L : - : H , \
                   L  H  H : - : H , \
                   H  -  - : - : N ";
    }
    pin(ENL) {
        direction : internal;
        internal_node : "ENL";
    }
    pin(EN) {
        direction : input;
        capacitance : 0.017997;
        clock_gate_enable_pin : true;
        timing() {
            timing_type : setup_rising;
            cell_rise(scalar) {values( " 0.4 "); }
            cell_fall(scalar) {values( " 0.4 "); }
        }
    }
}

```

```

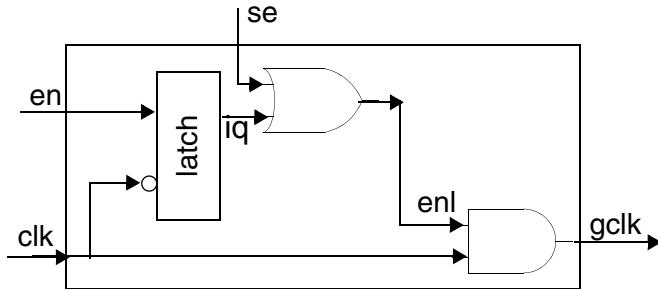
        related_pin : "CLK";
    }
    timing() {
        timing_type : hold_rising;
        cell_rise(scalar) {values( " 0.4 "); }
        cell_fall(scalar) {values( " 0.4 "); }
        related_pin : "CLK";
    }
}
pin(SE) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_test_pin : true;
}
pin(CLK) {
    direction : input;
    capacitance : 0.031419;
    clock_gate_clock_pin : true;
    min_pulse_width_low : 0.319;
}
pin(GCLK) {
    direction : output;
    state_function : "CLK * ENL";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
    timing() {
        timing_sense : positive_unate;
        cell_rise(scalar) {values( " 0.48 "); }
        cell_fall(scalar) {values( " 0.77 "); }
        rise_transition(scalar) {values( " 0.1443 "); }
        fall_transition(scalar) {values( " 0.0523 "); }
        related_pin : "CLK";
    }
}
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("0.141, 0.148, 0.256", \
        "0.162, 0.145, 0.234", \
        "0.192, 0.200, 0.284", \
        "0.199, 0.219, 0.297");
    }
    related_pin : "CLK EN" ;
}
}

```

latch_posedge_postcontrol Option

[Figure A-3](#) shows the circuit of the `latch_posedge_postcontrol` option for the `clock_gating_integrated_cell` attribute, and [Example A-3](#) demonstrates the use of this option.

Figure A-3 latch_posedge_postcontrol Circuit



Example A-3 latch_posedge_postcontrol Option

```

cell(CGLPC2) {
    area : 1;
    clock_gating_integrated_cell : "latch_posedge_postcontrol";
    dont_use : true;
    statetable(" CLK 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;
            cell_rise(scalar) {values( " 0.4 ");}
            cell_fall(scalar) {values( " 0.4 ");}
            related_pin : "CLK";
        }
        timing() {
            timing_type : hold_rising;
            cell_rise(scalar) {values( " 0.4 ");}
            cell_fall(scalar) {values( " 0.4 ");}
            related_pin : "CLK";
        }
    }
    pin(SE) {
        direction : input;
    }
}
  
```

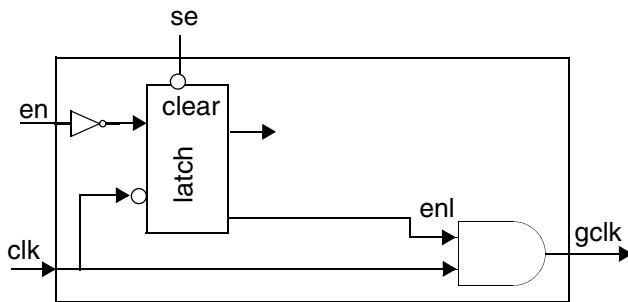
```

    capacitance : 0.017997;
    clock_gate_test_pin : true;
}
pin(CLK) {
    direction : input;
    capacitance : 0.031419;
    clock_gate_clock_pin : true;
    min_pulse_width_low : 0.319;
}
pin(GCLK) {
    direction : output;
    state_function : "CLK * (IQ + SE)";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
    timing() {
        timing_sense : positive_unate;
        cell_rise(scalar) {values( " 0.48 ");}
        cell_fall(scalar) {values( " 0.77 ");}
        rise_transition(scalar) {values( " 0.1443 ");}
        fall_transition(scalar) {values( " 0.0523 ");}
        related_pin : "CLK";
    }
    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("0.141, 0.148, 0.256", \
            "0.162, 0.145, 0.234", \
            "0.192, 0.200, 0.284", \
            "0.199, 0.219, 0.297");
        }
        related_pin : "CLK EN" ;
    }
}
}

```

[Figure A-4](#) shows the circuit of an inferred `latch_posedge_postcontrol` clock-gating integrated cell. The Library Compiler tool infers the `clock_gating_integrated_cell` attribute as a `latch_posedge_postcontrol` option and saves it to the .db file. [Example A-4](#) demonstrates the use of this option.

Figure A-4 Inferred latch_posedge_postcontrol Option Clock-Gating Integrated Cell



Example A-4 Inferred latch_posedge_postcontrol Option

```
cell(CGLPC2) {
    area : 1;
    clock_gating_integrated_cell : "generic";
    dont_use : true;
    latch ("IQ", "IQN") {
        enable : CLK';
        data_in : EN';
        clear : SE';
    }
    pin(EN) {
        direction : input;
        capacitance : 0.017997;
        clock_gate_enable_pin : true;
        timing() {
            rise_constraint(scalar) { values (" 0.4 "); }
            fall_constraint(scalar) { values (" 0.4 "); }
            timing_type : setup_rising;
            related_pin : "CLK";
        }
        timing() {
            rise_constraint(scalar) { values (" 0.4 "); }
            fall_constraint(scalar) { values (" 0.4 "); }
            timing_type : hold_rising;
            related_pin : "CLK";
        }
    }
    pin(SE) {
        direction : input;
        capacitance : 0.017997;
        clock_gate_test_pin : true;
    }
    pin(CLK) {
        direction : input;
        capacitance : 0.031419;
        clock_gate_clock_pin : true;
        min_pulse_width_low : 0.319;
    }
}
```

```

}
pin(GCLK) {
    direction : output;
    function : "CLK * IQN";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
    timing() {
        timing_sense : positive_unate;
        cell_rise(scalar) {values( " 0.48 ");}
        cell_fall(scalar) {values( " 0.77 ");}
        rise_transition(scalar) {values( " 0.1443 ");}
        fall_transition(scalar) {values( " 0.0523 ");}
        related_pin : "CLK";
    }
    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("0.141, 0.148, 0.256",
                   "0.162, 0.145, 0.234",
                   "0.192, 0.200, 0.284",
                   "0.199, 0.219, 0.297");
        }
        related_pin : "CLK EN" ;
    }
}
}

```

The Library Compiler tool automatically infers the `pin_is_active_high` attribute on the `clock_gate_test_pin` pin to indicate whether the pin is active high or active low. If the pin is active high, the attribute is set to `true`, or it is not specified at all; if the pin is active low, the attribute is set to `false`.

In addition, the Library Compiler tool infers the `clock_gate_reset_pin` attribute on the latch's clear pin to indicate whether the pin is a reset pin. If the pin is a reset pin for the latch, the attribute is automatically set to `true`; if the pin is not a reset pin for the latch, the attribute is not specified at all.

Important:

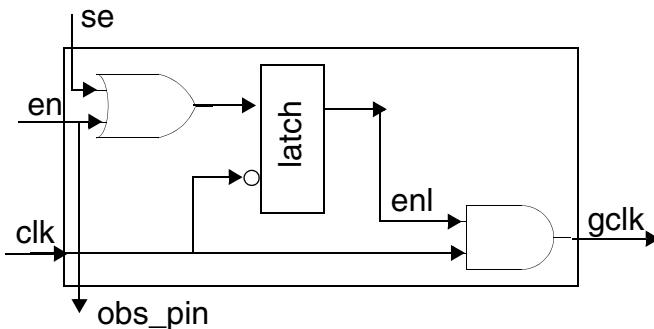
If you set the `pin_is_active_high` attribute to indicate whether a pin is active high or active low, or if you set the `clock_gate_reset_pin` attribute to indicate whether a pin is a reset pin for a `latch_posedge_postcontrol` generic integrated clock gating cell, the

Library Compiler tool issues an error message. The Library Compiler tool infers the values based on the pins. You cannot specify the values.

latch_posedge_precontrol_obs Option

[Figure A-5](#) shows the circuit of the `latch_posedge_precontrol_obs` option for the `clock_gating_integrated_cell` attribute, and [Example A-5](#) demonstrates the use of this option.

Figure A-5 latch_posedge_precontrol_obs Option Circuit



Example A-5 latch_posedge_precontrol_obs Option

```

cell(CGLPCO) {
    area : 1;
    clock_gating_integrated_cell : "latch_posedge_precontrol_obs";
    dont_use : true;
    statetable(" CLK EN SE", "ENL") {
        table : " L  L  L : - : L , \
                  L  L  H : - : H , \
                  L  H  L : - : H , \
                  L  H  H : - : H , \
                  H  -  - : - : N ";
    }
    pin(ENL) {
        direction : internal;
        internal_node : "ENL";
    }
    pin(EN) {
        direction : input;
        capacitance : 0.017997;
        clock_gate_enable_pin : true;
        timing() {
            timing_type : setup_rising;
            cell_rise(scalar) {values( " 0.4 ");}
            cell_fall(scalar) {values( " 0.4 ");}
            related_pin : "CLK";
        }
        timing() {
            timing_type : hold_rising;
            cell_rise(scalar) {values( " 0.4 ");}
        }
    }
}

```

```

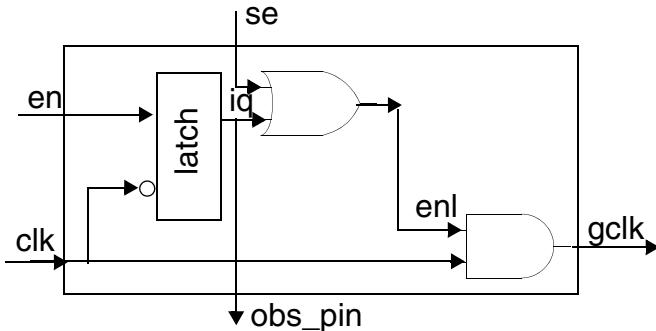
        cell_fall(scalar) {values( " 0.4 "); }
        related_pin : "CLK";
    }
}
pin(SE) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_test_pin : true;
}
pin(CLK) {
    direction : input;
    capacitance : 0.031419;
    clock_gate_clock_pin : true;
    min_pulse_width_low : 0.319;
}
pin(GCLK) {
    direction : output;
    state_function : "CLK * ENL";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
    timing() {
        timing_sense : positive_unate;
        cell_rise(scalar) {values( " 0.48 "); }
        cell_fall(scalar) {values( " 0.77 "); }
        rise_transition(scalar) {values( " 0.1443 "); }
        fall_transition(scalar) {values( " 0.0523 "); }
        related_pin : "EN CLK";
    }
    internal_power () {
        rise_power(l14X3) {
            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(l14X3) {
            index_1("0.0150, 0.0400, 0.1050, 0.3550");
            index_2("0.050, 0.451, 1.500");
            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");
        }
        related_pin : "EN" ;
    }
}
pin(OBS_PIN) {
    direction : output;
    state_function : "EN";
    max_capacitance : 0.500;
    clock_gate_obs_pin : true;
}
}

```

latch_posedge_postcontrol_obs Option

[Figure A-6](#) shows the circuit of the `latch_posedge_postcontrol_obs` option for the `clock_gating_integrated_cell` attribute, and [Example A-6](#) demonstrates the use of this option.

Figure A-6 latch_posedge_postcontrol_obs Circuit



Example A-6 latch_posedge_postcontrol_obs Option

```
cell(CGLPC20) {
    area : 1;
    clock_gating_integrated_cell : "latch_posedge_postcontrol_obs";
    dont_use : true;
    statetable(" CLK EN ", "IQ") {
        table : " L  L  : -  : L , \
                  L  H  : -  : H , \
                  H  -  : -  : N ";
    }
    pin(EN) {
        direction : input;
        capacitance : 0.017997;
        clock_gate_enable_pin : true;
        timing() {
            timing_type : setup_rising;
            cell_rise(scalar) {values( " 0.4 ");}
            cell_fall(scalar) {values( " 0.4 ");}
            related_pin : "CLK";
        }
        timing() {
            timing_type : hold_rising;
            cell_rise(scalar) {values( " 0.4 ");}
            cell_fall(scalar) {values( " 0.4 ");}
            related_pin : "CLK";
        }
    }
    pin(SE) {
        direction : input;
        capacitance : 0.017997;
        clock_gate_test_pin : true;
    }
    pin(CLK) {
```

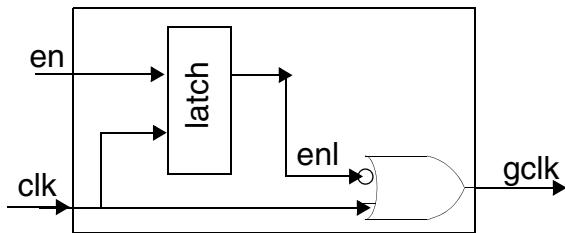
```

direction : input;
capacitance : 0.031419;
clock_gate_clock_pin : true;
min_pulse_width_low : 0.319;
}
pin(IQ) {
  direction : internal;
  internal_node : "IQ";
}
pin(GCLK) {
  direction : output;
  state_function : "CLK * (IQ + SE)";
  max_capacitance : 0.500;
  clock_gate_out_pin : true;
  timing() {
    timing_sense : positive_unate;
    cell_rise(scalar) {values( " 0.48 ");}
    cell_fall(scalar) {values( " 0.77 ");}
    rise_transition(scalar) {values( " 0.1443 ");}
    fall_transition(scalar) {values( " 0.0523 ");}
    related_pin : "CLK";
  }
  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("0.141, 0.148, 0.256", \
      "0.162, 0.145, 0.234", \
      "0.192, 0.200, 0.284", \
      "0.199, 0.219, 0.297");
    }
    related_pin : "CLK EN" ;
  }
  pin(OBS_PIN) {
    direction : output;
    internal_node : "IQ";
    max_capacitance : 0.500;
    clock_gate_obs_pin : true;
  }
}

```

latch_negedge Option

Figure A-7 shows the circuit of the `latch_negedge` option for the `clock_gating_integrated_cell` attribute, and Example A-7 demonstrates the use of this option.

Figure A-7 latch_negedge Circuit**Example A-7 latch_negedge Option**

```
cell(CGLN) {
    area : 1;
    clock_gating_integrated_cell : "latch_negedge";
    dont_use : true;
    dont_touch : true;
    statetable(" CLK EN ", "ENL ") {
        table : " H L : - : L , \
                  H H : - : H , \
                  L - : - : N ";
    }
    pin(ENL) {
        direction : internal;
        internal_node : "ENL";
    }
    pin(EN) {
        direction : input;
        capacitance : 0.017997;
        clock_gate_enable_pin : true;
        timing() {
            timing_type : setup_falling;
            cell_rise(scalar) {values( " 0.4 ");}
            cell_fall(scalar) {values( " 0.4 ");}
            related_pin : "CLK";
        }
        timing() {
            timing_type : hold_falling;
            cell_rise(scalar) {values( " 0.4 ");}
            cell_fall(scalar) {values( " 0.4 ");}
            related_pin : "CLK";
        }
    }
    pin(CLK) {
        direction : input;
        capacitance : 0.031419;
        clock_gate_clock_pin : true;
        min_pulse_width_low : 0.319;
    }
    pin(GCLK) {
        direction : output;
        state_function : "CLK + ENL'";
        max_capacitance : 0.500;
    }
}
```

```

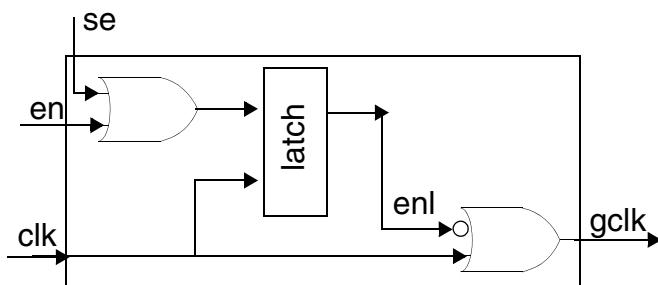
clock_gate_out_pin : true;
timing() {
    timing_sense : positive_unate;
    cell_rise(scalar) {values( " 0.48 ");}
    cell_fall(scalar) {values( " 0.77 ");}
    rise_transition(scalar) {values( " 0.1443 ");}
    fall_transition(scalar) {values( " 0.0523 ");}
    related_pin : "EN CLK";
}
internal_power () {
    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 : "EN" ;
}
}
}
}

```

latch_negedge_precontrol Option

[Figure A-8](#) shows the circuit of the `latch_negedge_precontrol` option for the `clock_gating_integrated_cell` attribute, and [Example A-8](#) demonstrates the use of this option.

Figure A-8 latch_negedge_precontrol Circuit



Example A-8 latch_negedge_precontrol Option

```

cell(CGLNC) {
    area : 1;
    clock_gating_integrated_cell : "latch_negedge_precontrol";
    dont_use : true;
    dont_touch : true;
    statetable(" CLK EN SE", "ENL ") {
        table : " H  L  L : - : L , \
                  H  L  H : - : H , \
                  H  H  L : - : H , \
                  H  H  H : - : H , \
                  L  -  - : - : N ";
    }
    pin(ENL) {
        direction : internal;
        internal_node : "ENL";
    }
    pin(EN) {
        direction : input;
        capacitance : 0.017997;
        clock_gate_enable_pin : true;
        timing() {
            timing_type : setup_falling;
            cell_rise(scalar) {values( " 0.4 ");}
            cell_fall(scalar) {values( " 0.4 ");}
            related_pin : "CLK";
        }
        timing() {
            timing_type : hold_falling;
            cell_rise(scalar) {values( " 0.4 ");}
            cell_fall(scalar) {values( " 0.4 ");}
            related_pin : "CLK";
        }
    }
    pin(SE) {
        direction : input;
        capacitance : 0.017997;
        clock_gate_test_pin : true;
    }
    pin(CLK) {
        direction : input;
        capacitance : 0.031419;
        clock_gate_clock_pin : true;
        min_pulse_width_low : 0.319;
    }
    pin(GCLK) {
        direction : output;
        state_function : "CLK + ENL'";
        max_capacitance : 0.500;
        clock_gate_out_pin : true;
        timing() {
            timing_sense : positive_unate;
            cell_rise(scalar) {values( " 0.48 ");}
            cell_fall(scalar) {values( " 0.77 ");}
            rise_transition(scalar) {values( " 0.1443 ");}
            fall_transition(scalar) {values( " 0.0523 ");}
            related_pin : "CLK";
        }
    }
}

```

```

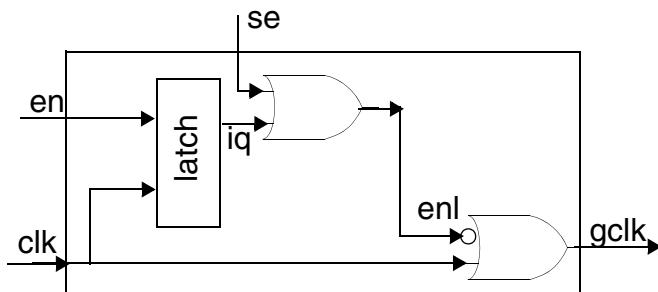
    }
internal_power () {
    rise_power(li4X3) {
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.501");
        values("0.141, 0.148, 0.256",
               "0.162, 0.145, 0.234",
               "0.192, 0.200, 0.284",
               "0.199, 0.219, 0.297");
    }
    fall_power(li4X3) {
        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 : "CLK EN" ;
}
}

```

latch_nedge_postcontrol Option

[Figure A-9](#) shows the circuit of the `latch_nedge_postcontrol` option for the `clock_gating_integrated_cell` attribute, and [Example A-9](#) demonstrates the use of this option.

Figure A-9 latch_nedge_postcontrol Circuit



Example A-9 latch negedge postcontrol Option

```
cell(CGLNC2) {
    area : 1;
    clock_gating_integrated_cell : "latch_nedgedge_postcontrol";
    dont_use : true;
    dont_touch : true;
```

```

statetable(" CLK EN ", "IQ ") {
    table : " H  L  : -  : L , \
              H  H  : -  : H , \
              L  -  : -  : N ";
}
pin(IQ) {
    direction : internal;
    internal_node : "IQ";
}
pin(EN) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_enable_pin : true;
    timing() {
        timing_type : setup_falling;
        cell_rise(scalar) {values( " 0.4 ");}
        cell_fall(scalar) {values( " 0.4 ");}
        related_pin : "CLK";
    }
    timing() {
        timing_type : hold_falling;
        cell_rise(scalar) {values( " 0.4 ");}
        cell_fall(scalar) {values( " 0.4 ");}
        related_pin : "CLK";
    }
}
pin(SE) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_test_pin : true;
}
pin(CLK) {
    direction : input;
    capacitance : 0.031419;
    clock_gate_clock_pin : true;
    min_pulse_width_low : 0.319;
}
pin(GCLK) {
    direction : output;
    state_function : "CLK + !(IQ + SE)";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
    timing() {
        timing_sense : positive_unate;
        cell_rise(scalar) {values( " 0.48 ");}
        cell_fall(scalar) {values( " 0.77 ");}
        rise_transition(scalar) {values( " 0.1443 ");}
        fall_transition(scalar) {values( " 0.0523 ");}
        related_pin : "CLK";
    }
    internal_power (){
        rise_power(l14X3){
            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");
        }
    }
}

```

```

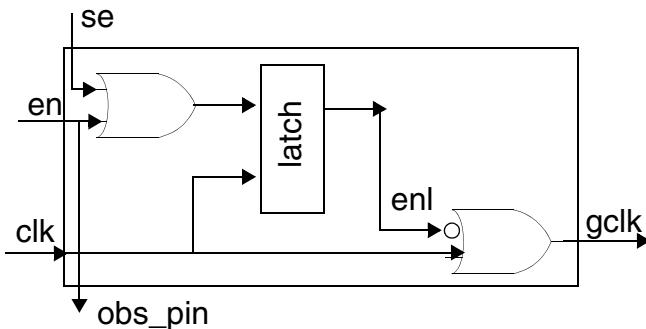
    }
    fall_power(lt4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.500");
        values("0.117, 0.144, 0.246",
               "0.133, 0.151, 0.238",
               "0.151, 0.186, 0.279",
               "0.160, 0.190, 0.217");
    }
    related_pin : "CLK EN" ;
}
}

```

latch_negedge_precontrol_obs Option

[Figure A-10](#) shows the circuit of the `latch_nedge_precontrol_obs` option for the `clock_gating_integrated_cell` attribute, and [Example A-10](#) demonstrates the use of this option.

Figure A-10 latch_nedge_precontrol_obs Circuit



Example A-10 latch_nedge_precontrol_obs Option

```

cell(CGLNCO) {
    area : 1;
    clock_gating_integrated_cell : "latch_nedgedge_precontrol_obs";
    dont_use : true;
    dont_touch : true;
    statetable(" CLK EN SE", "ENL ") {
        table : " H   L   L : - : L , \
                  H   L   H : - : H , \
                  H   H   L : - : H , \
                  H   H   H : - : H , \
                  L   -   - : - : N ";
    }
    pin(ENL) {
        direction : internal;
        internal_node : "ENL";
    }
    pin(EN) {
        direction : input;
    }
}

```

```

capacitance : 0.017997;
clock_gate_enable_pin : true;
timing() {
    timing_type : setup_falling;
    cell_rise(scalar) {values( " 0.4 "); }
    cell_fall(scalar) {values( " 0.4 "); }
    related_pin : "CLK";
}
timing() {
    timing_type : hold_falling;
    cell_rise(scalar) {values( " 0.4 "); }
    cell_fall(scalar) {values( " 0.4 "); }
    related_pin : "CLK";
}
pin(SE) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_test_pin : true;
}
pin(CLK) {
    direction : input;
    capacitance : 0.031419;
    clock_gate_clock_pin : true;
    min_pulse_width_low : 0.319;
}
pin(GCLK) {
    direction : output;
    state_function : "CLK + ENL'";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
    timing() {
        timing_sense : positive_unate;
        cell_rise(scalar) {values( " 0.48 "); }
        cell_fall(scalar) {values( " 0.77 "); }
        rise_transition(scalar) {values( " 0.1443 "); }
        fall_transition(scalar) {values( " 0.0523 "); }
        related_pin : "CLK";
    }
    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 : "CLK EN" ;
    }
}

```

```

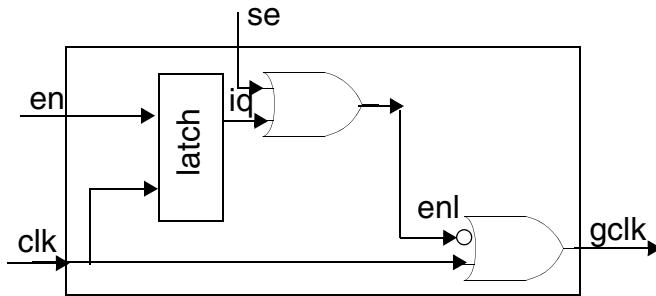
    }
    pin(OBS_PIN) {
        direction : output;
        state_function : "EN";
        max_capacitance : 0.500;
        clock_gate_obs_pin : true;
    }
}

```

latch_nedgede_postcontrol_obs Option

[Figure A-11](#) shows the circuit of the `latch_nedgede_postcontrol_obs` option for the `clock_gating_integrated_cell` attribute, and [Example A-11](#) demonstrates the use of this option.

Figure A-11 latch_nedgede_postcontrol_obs Circuit



Example A-11 latch_nedgede_postcontrol_obs Option

```

cell(CGLNC20) {
    area : 1;
    clock_gating_integrated_cell : "latch_nedgede_postcontrol_obs";
    dont_use : true;
    dont_touch : true;
    statetable(" CLK EN ", "IQ") {
        table : " H L : - : L , \
                  H H : - : H , \
                  L - : - : N ";
    }
    pin(EN) {
        direction : input;
        capacitance : 0.017997;
        clock_gate_enable_pin : true;
        timing() {
            timing_type : setup_falling;
            cell_rise(scalar) {values( " 0.4 ");}
            cell_fall(scalar) {values( " 0.4 ");}
            related_pin : "CLK";
        }
    }
}

```

```

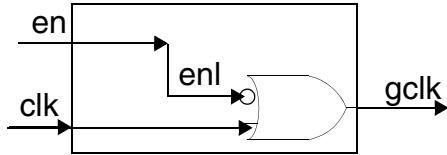
timing() {
    timing_type : hold_falling;
    cell_rise(scalar) {values( " 0.4 ");}
    cell_fall(scalar) {values( " 0.4 ");}
    related_pin : "CLK";
}
}
pin(SE) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_test_pin : true;
}
pin(CLK) {
    direction : input;
    capacitance : 0.031419;
    clock_gate_clock_pin : true;
    min_pulse_width_low : 0.319;
}
pin(IQ) {
    direction : internal;
    internal_node : "IQ";
}
pin(GCLK) {
    direction : output;
    state_function : "CLK + !(IQ + SE)";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
    timing() {
        timing_sense : positive_unate;
        cell_rise(scalar) {values( " 0.48 ");}
        cell_fall(scalar) {values( " 0.77 ");}
        rise_transition(scalar) {values( " 0.1443 ");}
        fall_transition(scalar) {values( " 0.0523 ");}
        related_pin : "CLK";
    }
    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 : "CLK EN" ;
    }
}
}

```

none_posedge Option

[Figure A-12](#) shows the circuit of the `none_posedge` option for the `clock_gating_integrated_cell` attribute, and [Example A-12](#) demonstrates the use of this option.

Figure A-12 none_posedge Circuit



Example A-12 none_posedge Option

```

cell(CGNP) {
    area : 1;
    clock_gating_integrated_cell : "none_posedge";
    dont_use : true;
    pin(EN) {
        direction : input;
        capacitance : 0.017997;
        clock_gate_enable_pin : true;
        timing() {
            timing_type : nochange_high_low;
            cell_rise(scalar) {values( " 0.4 ")} ;
            cell_fall(scalar) {values( " 0.4 ")} ;
            related_pin : "CLK";
        }
        timing() {
            timing_type : nochange_low_low;
            cell_rise(scalar) {values( " 0.4 ")} ;
            cell_fall(scalar) {values( " 0.4 ")} ;
            related_pin : "CLK";
        }
    }
    pin(CLK) {
        direction : input;
        capacitance : 0.031419;
        clock_gate_clock_pin : true;
        min_pulse_width_low : 0.319;
    }
    pin(GCLK) {
        direction : output;
        function : "CLK + EN'";
        max_capacitance : 0.500;
        clock_gate_out_pin : true;
        timing() {
            timing_sense : positive_unate;
            cell_rise(scalar) {values( " 0.48 ")} ;
        }
    }
}
  
```

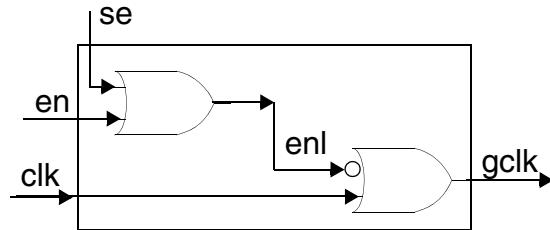
```

    cell_fall(scalar) {values( " 0.77 "); }
    rise_transition(scalar) {values( " 0.1443 "); }
    fall_transition(scalar) {values( " 0.0523 "); }
    related_pin : "EN";
}
timing() {
    timing_sense : positive_unate;
    cell_rise(scalar) {values( " 0.48 "); }
    cell_fall(scalar) {values( " 0.77 "); }
    rise_transition(scalar) {values( " 0.1443 "); }
    fall_transition(scalar) {values( " 0.0523 "); }
    related_pin : "CLK";
}
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 : "CLK EN" ;
}
}
}

```

none_posedge_control Option

[Figure A-13](#) shows the circuit of the `none_posedge_control` option for the `clock_gating_integrated_cell` attribute, and [Example A-13](#) demonstrates the use of this option.

Figure A-13 *none_posedge_control* Circuit**Example A-13** *none_posedge_control* Option

```
cell(CGNPC) {
    area : 1;
    clock_gating_integrated_cell : "none_posedge_control";
    dont_use : true;
    pin(EN) {
        direction : input;
        capacitance : 0.017997;
        clock_gate_enable_pin : true;
        timing() {
            timing_type : nochange_high_low;
            cell_rise(scalar) {values( " 0.4 ");}
            cell_fall(scalar) {values( " 0.4 ");}
            related_pin : "CLK";
        }
        timing() {
            timing_type : nochange_low_low;
            cell_rise(scalar) {values( " 0.4 ");}
            cell_fall(scalar) {values( " 0.4 ");}
            related_pin : "CLK";
        }
    }
    pin(SE) {
        direction : input;
        capacitance : 0.017997;
        clock_gate_test_pin : true;
    }
    pin(CLK) {
        direction : input;
        capacitance : 0.031419;
        clock_gate_clock_pin : true;
        min_pulse_width_low : 0.319;
    }
    pin(GCLK) {
        direction : output;
        function : "CLK + !(SE + EN)";
        max_capacitance : 0.500;
        clock_gate_out_pin : true;
        internal_power () {
            rise_power(l14X3) {
                index_1("0.0150, 0.0400, 0.1050, 0.3550");
            }
        }
    }
}
```

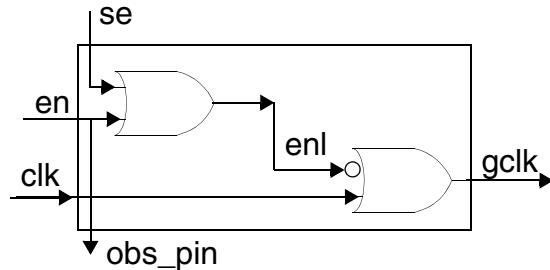
```

        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("0.141, 0.148, 0.256", \
"0.162, 0.145, 0.234", \
"0.192, 0.200, 0.284", \
"0.199, 0.219, 0.297");
    }
    related_pin : "CLK EN" ;
}
timing() {
    timing_sense : positive_unate;
    cell_rise(scalar) {values( " 0.48 "); }
    cell_fall(scalar) {values( " 0.77 "); }
    rise_transition(scalar) {values( " 0.1443 "); }
    fall_transition(scalar) {values( " 0.0523 "); }
    related_pin : "EN";
}
timing() {
    timing_sense : positive_unate;
    cell_rise(scalar) {values( " 0.48 "); }
    cell_fall(scalar) {values( " 0.77 "); }
    rise_transition(scalar) {values( " 0.1443 "); }
    fall_transition(scalar) {values( " 0.0523 "); }
    related_pin : "SE";
}
timing() {
    timing_sense : positive_unate;
    cell_rise(scalar) {values( " 0.48 "); }
    cell_fall(scalar) {values( " 0.77 "); }
    rise_transition(scalar) {values( " 0.1443 "); }
    fall_transition(scalar) {values( " 0.0523 "); }
    related_pin : "CLK";
}
}
}

```

none_posedge_control_obs Option

[Figure A-14](#) shows the circuit of the `none_posedge_control_obs` option for the `clock_gating_integrated_cell` attribute, and [Example A-14](#) demonstrates the use of this option.

Figure A-14 *none_posedge_control_obs* Circuit**Example A-14** *none_posedge_control_obs* Option

```
cell(CGNPCO) {
    area : 1;
    clock_gating_integrated_cell : "none_posedge_control_obs";
    dont_use : true;
    pin(EN) {
        direction : input;
        capacitance : 0.017997;
        clock_gate_enable_pin : true;
        timing() {
            timing_type : nochange_high_low;
            cell_rise(scalar) {values( " 0.4 ")}; }
            cell_fall(scalar) {values( " 0.4 ")}; }
            related_pin : "CLK";
    }
    timing() {
        timing_type : nochange_low_low;
        cell_rise(scalar) {values( " 0.4 ")}; }
        cell_fall(scalar) {values( " 0.4 ")}; }
        related_pin : "CLK";
    }
    pin(SE) {
        direction : input;
        capacitance : 0.017997;
        clock_gate_test_pin : true;
    }
    pin(CLK) {
        direction : input;
        capacitance : 0.031419;
        clock_gate_clock_pin : true;
        min_pulse_width_low : 0.319;
    }
    pin(OBS_PIN) {
        direction : output;
        function : "EN";
        clock_gate_obs_pin : true;
        max_capacitance : 0.500;
        timing() {
            timing_sense : positive_unate;
            cell_rise(scalar) {values( " 0.48 ")}; }
    }
}
```

```

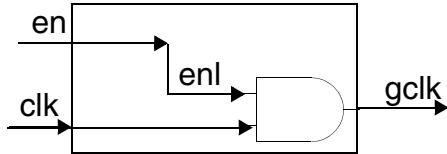
    cell_fall(scalar) {values( " 0.77 "); }
    rise_transition(scalar) {values( " 0.1443 "); }
    fall_transition(scalar) {values( " 0.0523 "); }
    related_pin : "EN";
}
}
pin(GCLK) {
    direction : output;
    function : "CLK + !(SE + EN)";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
    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("0.141, 0.148, 0.256", \
"0.162, 0.145, 0.234", \
"0.192, 0.200, 0.284", \
"0.199, 0.219, 0.297");
        }
        related_pin : "CLK EN" ;
    }
    timing() {
        timing_sense : positive_unate;
        cell_rise(scalar) {values( " 0.48 "); }
        cell_fall(scalar) {values( " 0.77 "); }
        rise_transition(scalar) {values( " 0.1443 "); }
        fall_transition(scalar) {values( " 0.0523 "); }
        related_pin : "EN";
    }
    timing() {
        timing_sense : positive_unate;
        cell_rise(scalar) {values( " 0.48 "); }
        cell_fall(scalar) {values( " 0.77 "); }
        rise_transition(scalar) {values( " 0.1443 "); }
        fall_transition(scalar) {values( " 0.0523 "); }
        related_pin : "SE";
    }
    timing() {
        timing_sense : positive_unate;
        cell_rise(scalar) {values( " 0.48 "); }
        cell_fall(scalar) {values( " 0.77 "); }
        rise_transition(scalar) {values( " 0.1443 "); }
        fall_transition(scalar) {values( " 0.0523 "); }
        related_pin : "CLK";
    }
}
}
}

```

none_nededge Option

[Figure A-15](#) shows the circuit of the `none_nededge` option for the `clock_gating_integrated_cell` attribute, and [Example A-15](#) demonstrates the use of this option.

Figure A-15 none_nededge Circuit



Example A-15 none_nededge Option

```

cell(CGNN) {
    area : 1;
    clock_gating_integrated_cell : "none_nededge";
    dont_use : true;
    dont_touch : true;
    pin(EN) {
        direction : input;
        capacitance : 0.017997;
        clock_gate_enable_pin : true;
        timing() {
            timing_type : nochange_high_high;
            cell_rise(scalar) {values( " 0.4 ");}
            cell_fall(scalar) {values( " 0.4 ");}
            related_pin : "CLK";
        }
        timing() {
            timing_type : nochange_low_high;
            cell_rise(scalar) {values( " 0.4 ");}
            cell_fall(scalar) {values( " 0.4 ");}
            related_pin : "CLK";
        }
    }
    pin(CLK) {
        direction : input;
        capacitance : 0.031419;
        clock_gate_clock_pin : true;
        min_pulse_width_low : 0.319;
    }
    pin(GCLK) {
        direction : output;
        function : "CLK * EN";
        max_capacitance : 0.500;
        clock_gate_out_pin : true;
        internal_power () {
            rise_power(l14X3) {
                index_1("0.0150, 0.0400, 0.1050, 0.3550");
                index_2("0.050, 0.451, 1.501");
            }
        }
    }
}
  
```

```

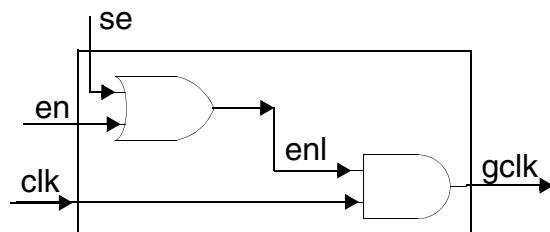
        values("0.141, 0.148, 0.256", \
"0.162, 0.145, 0.234", \
"0.192, 0.200, 0.284", \
"0.199, 0.219, 0.297");
    }
    fall_power(li4X3){
        index_1("0.0150, 0.0400, 0.1050, 0.3550");
        index_2("0.050, 0.451, 1.500");
        values("0.117, 0.144, 0.246", \
"0.133, 0.151, 0.238", \
"0.151, 0.186, 0.279", \
"0.160, 0.190, 0.217");
    }
    related_pin : "CLK EN" ;
}
timing() {
    timing_sense : positive_unate;
    cell_rise(scalar) {values( " 0.48 "); }
    cell_fall(scalar) {values( " 0.77 "); }
    rise_transition(scalar) {values( " 0.1443 "); }
    fall_transition(scalar) {values( " 0.0523 "); }
    related_pin : "EN";
}
timing() {
    timing_sense : positive_unate;
    cell_rise(scalar) {values( " 0.48 "); }
    cell_fall(scalar) {values( " 0.77 "); }
    rise_transition(scalar) {values( " 0.1443 "); }
    fall_transition(scalar) {values( " 0.0523 "); }
    related_pin : "CLK";
}
}
}
}

```

none_nededge_control Option

[Figure A-16](#) shows the circuit of the `none_nededge_control` option for the `clock_gating_integrated_cell` attribute, and [Example A-16](#) demonstrates the use of this option.

Figure A-16 none_nededge_control Circuit



Example A-16 none_nededge_control Option

```

cell(CGNNC) {
    area : 1;
    clock_gating_integrated_cell : "none_nededge_control";
    dont_use : true;
    dont_touch : true;
    pin(EN) {
        direction : input;
        capacitance : 0.017997;
        clock_gate_enable_pin : true;
        timing() {
            timing_type : nochange_high_high;
            cell_rise(scalar) {values( " 0.4 ");}
            cell_fall(scalar) {values( " 0.4 ");}
            related_pin : "CLK";
        }
        timing() {
            timing_type : nochange_low_high;
            cell_rise(scalar) {values( " 0.4 ");}
            cell_fall(scalar) {values( " 0.4 ");}
            related_pin : "CLK";
        }
    }
    pin(SE) {
        direction : input;
        capacitance : 0.017997;
        clock_gate_test_pin : true;
    }
    pin(CLK) {
        direction : input;
        capacitance : 0.031419;
        clock_gate_clock_pin : true;
        min_pulse_width_low : 0.319;
    }

    pin(GCLK) {
        direction : output;
        function : "CLK * (SE + EN)";
        max_capacitance : 0.500;
        clock_gate_out_pin : true;
        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");
            }
        }
    }
}

```

```

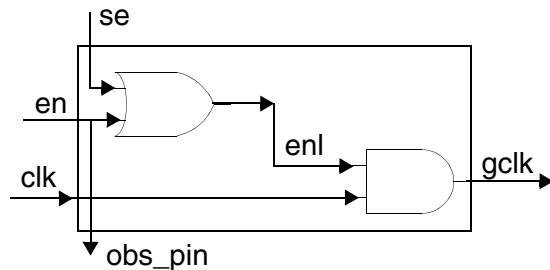
        related_pin : "CLK EN" ;
    }
    timing() {
        timing_sense : positive_unate;
        cell_rise(scalar) {values( " 0.48 ")} ;
        cell_fall(scalar) {values( " 0.77 ")} ;
        rise_transition(scalar) {values( " 0.1443 ")} ;
        fall_transition(scalar) {values( " 0.0523 ")} ;
        related_pin : "EN";
    }
    timing() {
        timing_sense : positive_unate;
        cell_rise(scalar) {values( " 0.48 ")} ;
        cell_fall(scalar) {values( " 0.77 ")} ;
        rise_transition(scalar) {values( " 0.1443 ")} ;
        fall_transition(scalar) {values( " 0.0523 ")} ;
        related_pin : "SE";
    }
    timing() {
        timing_sense : positive_unate;
        cell_rise(scalar) {values( " 0.48 ")} ;
        cell_fall(scalar) {values( " 0.77 ")} ;
        rise_transition(scalar) {values( " 0.1443 ")} ;
        fall_transition(scalar) {values( " 0.0523 ")} ;
        related_pin : "CLK";
    }
}
}
}

```

none_nededge_control_obs Option

[Figure A-17](#) shows the circuit of the `none_nededge_control_obs` option for the `clock_gating_integrated_cell` attribute, and [Example A-17](#) demonstrates the use of this option.

Figure A-17 none_nededge_control_obs Circuit



Example A-17 none_nededge_control_obs Option

```

cell(CGNNCO) {
    area : 1;
}

```

```

clock_gating_integrated_cell : "none_nededge_control_obs";
dont_use : true;
dont_touch : true;
pin(EN) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_enable_pin : true;
    timing() {
        timing_type : nochange_high_high;
        cell_rise(scalar) {values( " 0.4 "); }
        cell_fall(scalar) {values( " 0.4 "); }
        related_pin : "CLK";
    }
    timing() {
        timing_type : nochange_low_high;
        cell_rise(scalar) {values( " 0.4 "); }
        cell_fall(scalar) {values( " 0.4 "); }
        related_pin : "CLK";
    }
}
pin(SE) {
    direction : input;
    capacitance : 0.017997;
    clock_gate_test_pin : true;
}
pin(CLK) {
    direction : input;
    capacitance : 0.031419;
    clock_gate_clock_pin : true;
    min_pulse_width_low : 0.319;
}
pin(OBS_PIN) {
    direction : output;
    function : "EN";
    clock_gate_obs_pin : true;
    max_capacitance : 0.500;
    timing() {
        timing_sense : positive_unate;
        cell_rise(scalar) {values( " 0.48 "); }
        cell_fall(scalar) {values( " 0.77 "); }
        rise_transition(scalar) {values( " 0.1443 "); }
        fall_transition(scalar) {values( " 0.0523 "); }
        related_pin : "EN";
    }
}
pin(GCLK) {
    direction : output;
    function : "CLK * (SE + EN)";
    max_capacitance : 0.500;
    clock_gate_out_pin : true;
    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 : "CLK EN" ;
}
timing() {
    timing_sense : positive_unate;
    cell_rise(scalar) {values( " 0.48 ")} ;
    cell_fall(scalar) {values( " 0.77 ")} ;
    rise_transition(scalar) {values( " 0.1443 ")} ;
    fall_transition(scalar) {values( " 0.0523 ")} ;
    related_pin : "EN";
}
timing() {
    timing_sense : positive_unate;
    cell_rise(scalar) {values( " 0.48 ")} ;
    cell_fall(scalar) {values( " 0.77 ")} ;
    rise_transition(scalar) {values( " 0.1443 ")} ;
    fall_transition(scalar) {values( " 0.0523 ")} ;
    related_pin : "SE";
}
timing() {
    timing_sense : positive_unate;
    cell_rise(scalar) {values( " 0.48 ")} ;
    cell_fall(scalar) {values( " 0.77 ")} ;
    rise_transition(scalar) {values( " 0.1443 ")} ;
    fall_transition(scalar) {values( " 0.0523 ")} ;
    related_pin : "CLK";
}
}
}
}

```

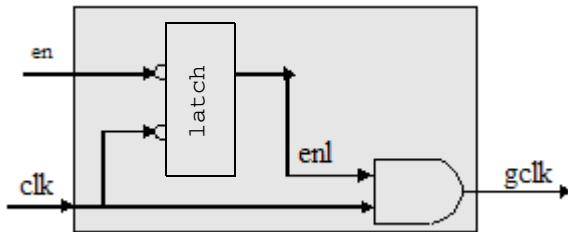
Generic Integrated Clock-Gating Schematics

This section provides circuit schematics for the cell structures listed in [Table A-2 on page A-5](#). After you model a cell using the generic integrated clock-gating syntax, the Library Compiler tool derives one of the following structures automatically.

latch_posedgeactivelow

[Figure A-18](#) shows the derived `latch_posedgeactivelow` integrated clock-gating cell circuit.

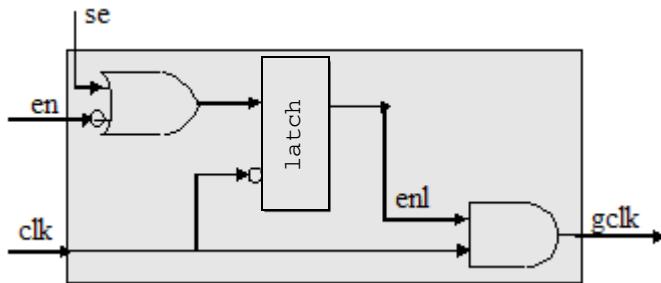
Figure A-18 latch_posedgeactiveelow Circuit



latch_posedgeactiveelow_precontrol

Figure A-19 shows the derived `latch_posedgeactiveelow_precontrol` integrated clock-gating cell circuit.

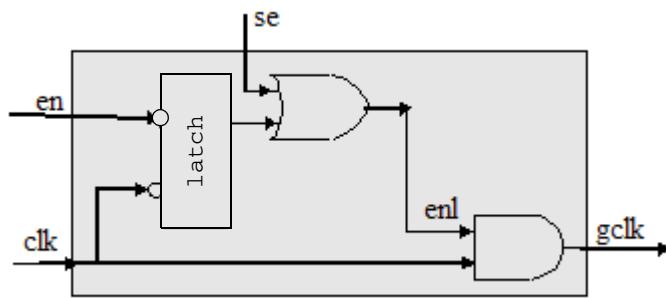
Figure A-19 latch_posedgeactiveelow_precontrol Circuit



latch_posedgeactiveelow_postcontrol

Figure A-20 shows the derived `latch_posedgeactiveelow_postcontrol` integrated clock-gating cell circuit.

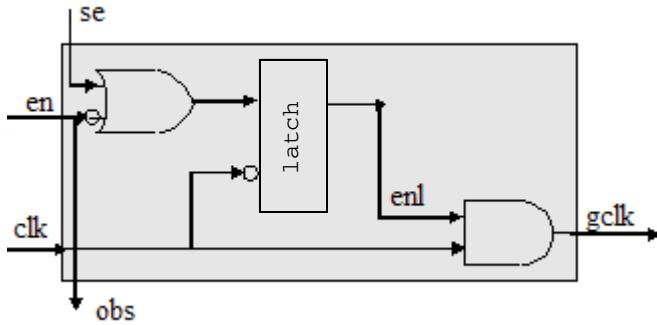
Figure A-20 latch_posedgeactiveelow_postcontrol Circuit



latch_posedgeactivelow_precontrol_obs

Figure A-21 shows the derived `latch_posedgeactivelow_precontrol_obs` integrated clock-gating cell circuit.

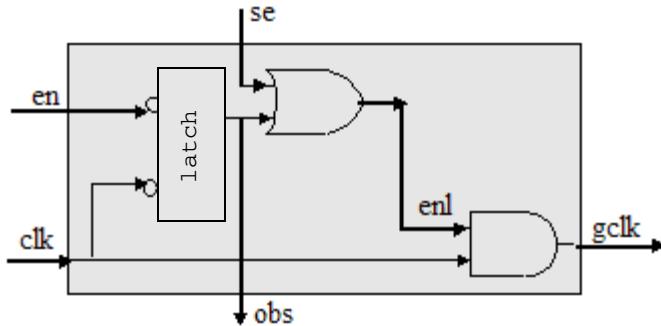
Figure A-21 *latch_posedgeactivelow_precontrol_obs Circuit*



latch_posedgeactivelow_postcontrol_obs

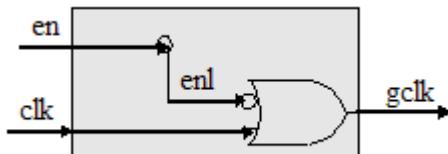
Figure A-22 shows the derived `latch_posedgeactivelow_postcontrol_obs` integrated clock-gating cell circuit.

Figure A-22 *latch_posedgeactivelow_postcontrol_obs Circuit*

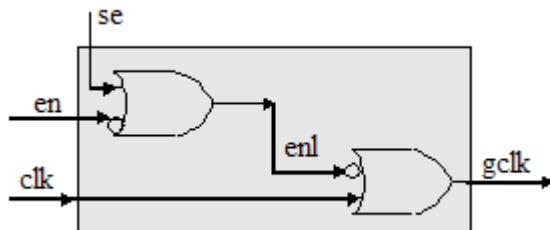


none_posedgeactivelow

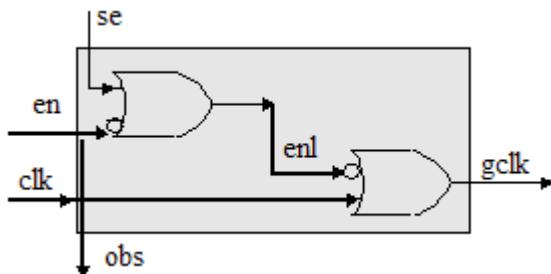
Figure A-23 shows the derived `none_posedgeactivelow` integrated clock-gating cell circuit.

Figure A-23 none_posedgeactivehigh Circuit**none_posedgeactivehigh_control**

[Figure A-24](#) shows the derived `none_posedgeactivehigh_control` integrated clock-gating cell circuit.

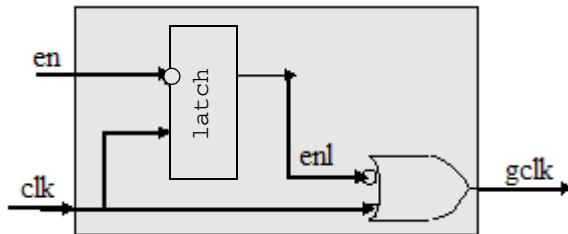
Figure A-24 none_posedgeactivehigh_control Circuit**none_posedgeactivehigh_control_obs**

[Figure A-25](#) shows the derived `none_posedgeactivehigh_control_obs` integrated clock-gating cell circuit.

Figure A-25 none_posedgeactivehigh_control_obs Circuit**latch_negedgeactivehigh**

[Figure A-26](#) shows the derived `latch_negedgeactivehigh` integrated clock-gating cell circuit.

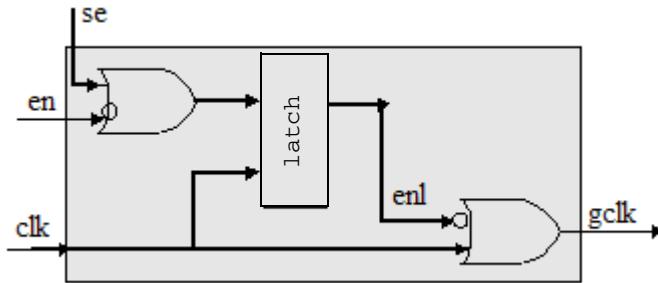
Figure A-26 latch_nedgedeactivelow Circuit



latch_nedgedeactivelow_precontrol

[Figure A-27](#) shows the derived `latch_nedgedeactivelow_precontrol` integrated clock-gating cell circuit.

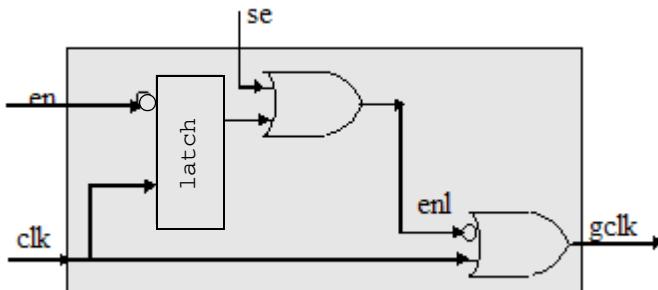
Figure A-27 latch_nedgedeactivelow_precontrol Circuit



latch_nedgedeactivelow_postcontrol

[Figure A-28](#) shows the derived `latch_nedgedeactivelow_postcontrol` integrated clock-gating cell circuit.

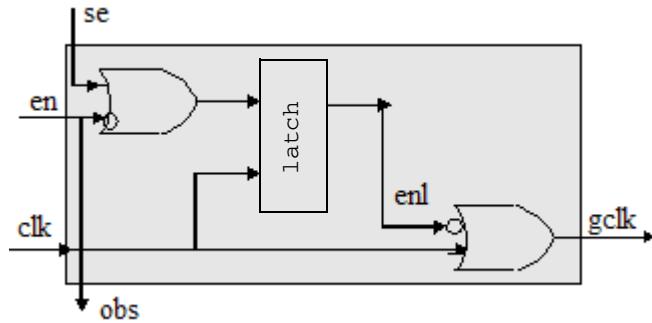
Figure A-28 latch_nedgedeactivelow_postcontrol Circuit



latch_nedgedeactivelow_precontrol_obs

[Figure A-29](#) shows the derived `latch_nedgedeactivelow_precontrol_obs` integrated clock-gating cell circuit.

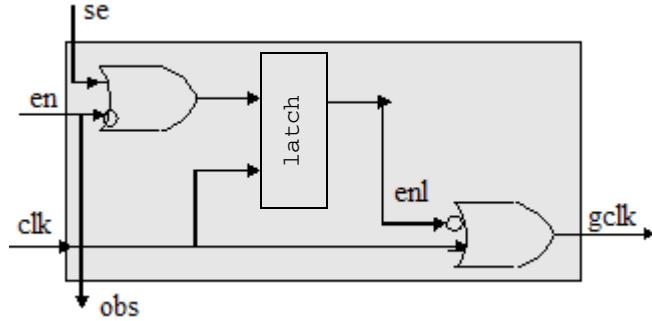
Figure A-29 latch_nedgedeactivelow_precontrol_obs Circuit



latch_nedgedeactivelow_postcontrol_obs

[Figure A-30](#) shows the derived `latch_nedgedeactivelow_postcontrol_obs` integrated clock-gating cell circuit.

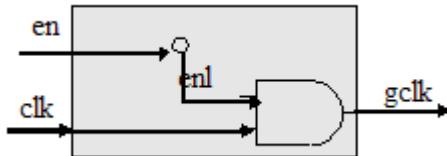
Figure A-30 latch_nedgedeactivelow_postcontrol_obs Circuit



none_nedgedeactivelow

[Figure A-31](#) shows the derived `none_nedgedeactivelow` integrated clock-gating cell circuit.

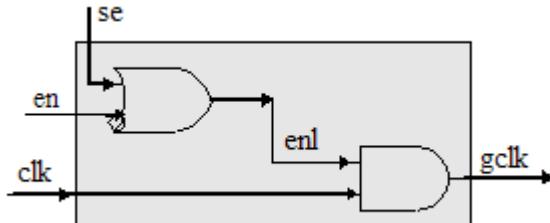
Figure A-31 none_nedgedeactivelow Circuit



none_nedgedetectivelow_control

Figure A-32 shows the derived `none_nedgedetectivelow_control` integrated clock-gating cell circuit.

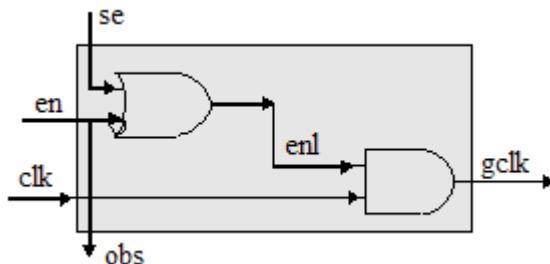
Figure A-32 `none_nedgedetectivelow_control` Circuit



none_nedgedetectivelow_control_obs

Figure A-33 shows the derived `none_nedgedetectivelow_control_obs` integrated clock-gating cell circuit.

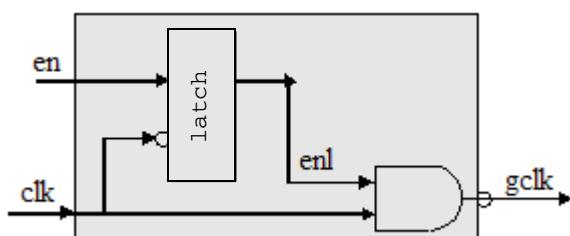
Figure A-33 `none_nedgedetectivelow_control_obs` Circuit



latch_posedge_invgclk

Figure A-34 shows the derived `latch_posedge_invgclk` integrated clock-gating cell circuit.

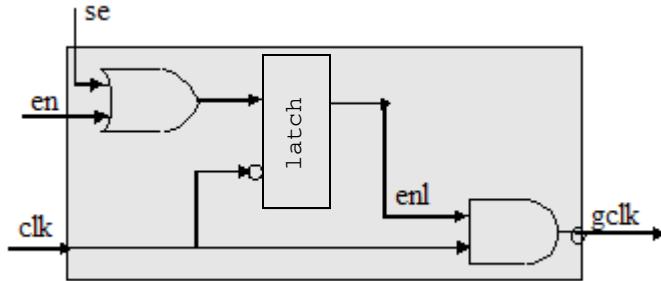
Figure A-34 `latch_posedge_invgclk` Circuit



latch_posedge_precontrol_invgclk

Figure A-35 shows the derived `latch_posedge_precontrol_invgclk` integrated clock-gating cell circuit.

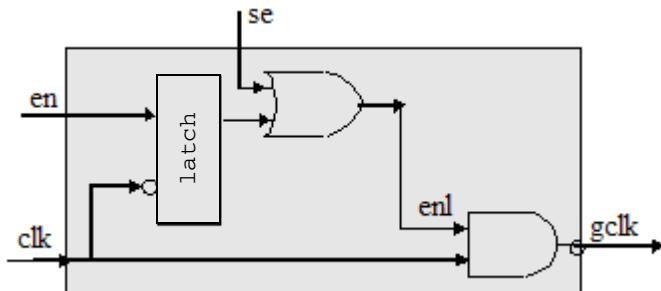
Figure A-35 latch_posedge_precontrol_invgclk Circuit



latch_posedge_postcontrol_invgclk

Figure A-36 shows the derived `latch_posedge_postcontrol_invgclk` integrated clock-gating cell circuit.

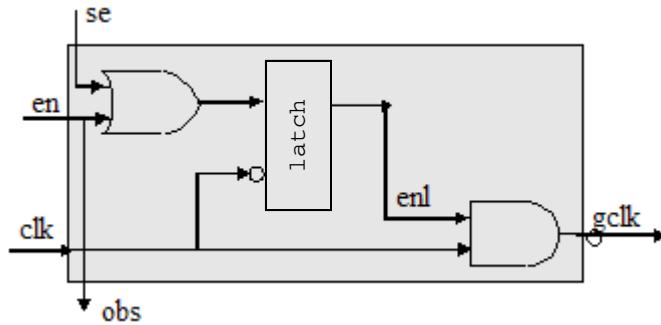
Figure A-36 latch_posedge_postcontrol_invgclk Circuit



latch_posedge_precontrol_obs_invgclk

Figure A-37 shows the derived `latch_posedge_precontrol_obs_invgclk` integrated clock-gating cell circuit.

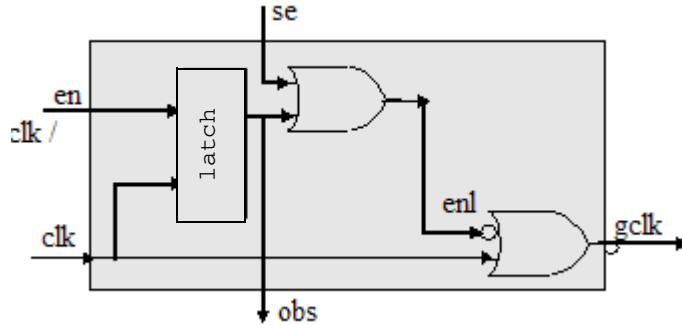
Figure A-37 latch_posedge_precontrol_obs_invclk Circuit



latch_posedge_postcontrol_obs_invclk

[Figure A-38](#) shows the derived `latch_posedge_postcontrol_obs_invclk` integrated clock-gating cell circuit.

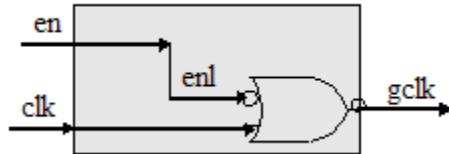
Figure A-38 latch_posedge_postcontrol_obs_invclk Circuit



none_posedge_invclk

[Figure A-39](#) shows the derived `none_posedge_invclk` integrated clock-gating cell circuit.

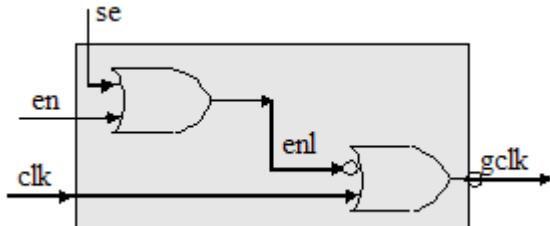
Figure A-39 none_posedge_invclk Circuit



none_posedge_control_invgclk

[Figure A-40](#) shows the derived `none_posedge_control_invgclk` integrated clock-gating cell circuit.

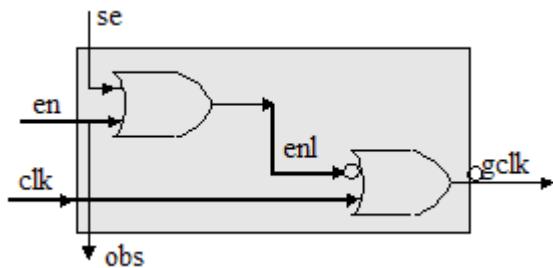
Figure A-40 none_posedge_control_invgclk Circuit



none_posedge_control_obs_invgclk

[Figure A-41](#) shows the derived `none_posedge_control_obs_invgclk` integrated clock-gating cell circuit.

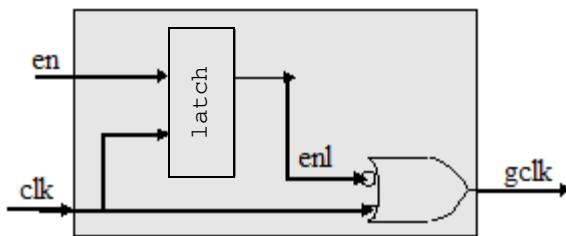
Figure A-41 none_posedge_control_obs_invgclk Circuit



latch_negedge_invgclk

[Figure A-42](#) shows the derived `latch_negedge_invgclk` integrated clock-gating cell circuit.

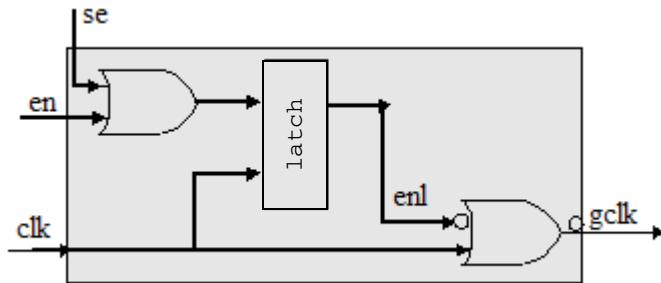
Figure A-42 latch_negedge_invgclk Circuit



latch_nededge_precontrol_invgclk

[Figure A-43](#) shows the derived `latch_nededge_precontrol_invgclk` integrated clock-gating cell circuit.

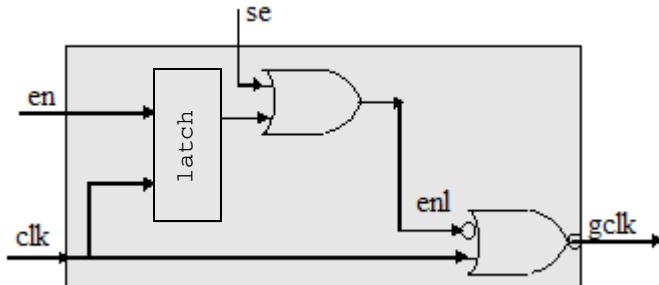
Figure A-43 latch_nededge_precontrol_invgclk Circuit



latch_nededge_postcontrol_invgclk

[Figure A-44](#) shows the derived `latch_nededge_postcontrol_invgclk` integrated clock-gating cell circuit.

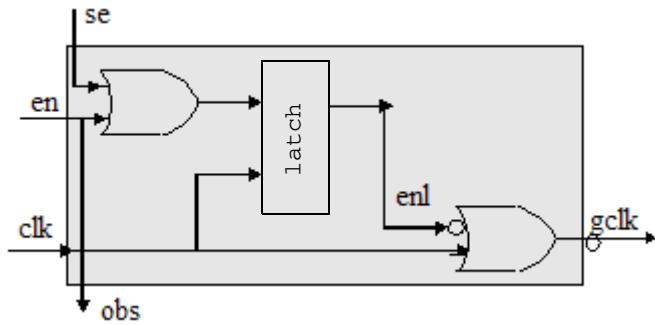
Figure A-44 latch_nededge_postcontrol_invgclk Circuit



latch_nededge_precontrol_obs_invgclk

[Figure A-45](#) shows the derived `latch_nededge_precontrol_obs_invgclk` integrated clock-gating cell circuit.

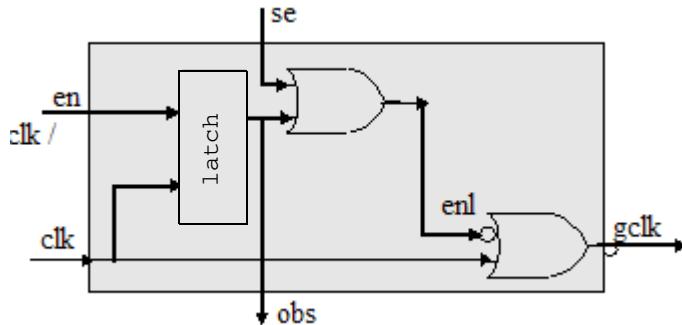
Figure A-45 latch_negedge_precontrol_obs_invclk Circuit



latch_negedge_postcontrol_obs_invclk

Figure A-46 shows the derived `latch_negedge_postcontrol_obs_invclk` integrated clock-gating cell circuit.

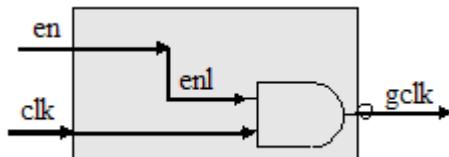
Figure A-46 latch_negedge_postcontrol_obs_invclk Circuit



none_negedge_invclk

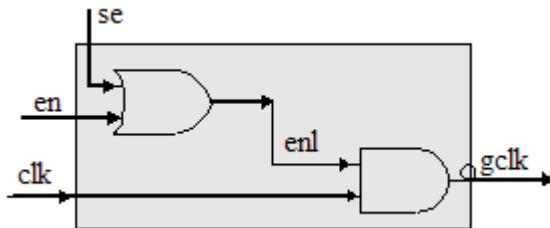
Figure A-47 shows the derived `none_negedge_invclk` integrated clock-gating cell circuit.

Figure A-47 none_negedge_invclk Circuit

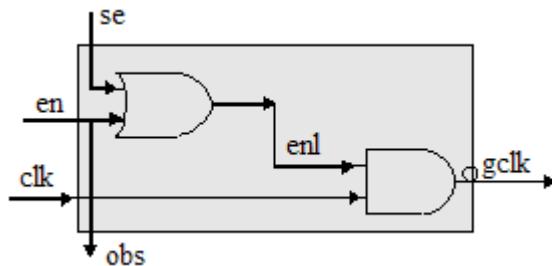


none_negedge_control_invclk

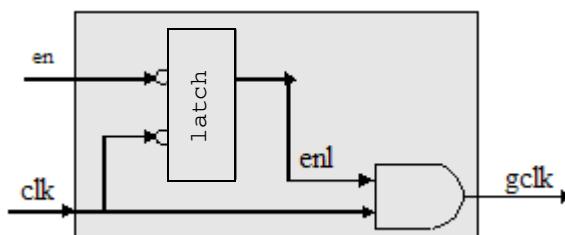
Figure A-48 shows the derived `none_negedge_control_invclk` integrated clock-gating cell circuit.

Figure A-48 none_nedgede_control_invgclk Circuit**none_nedgede_control_obs_invgclk**

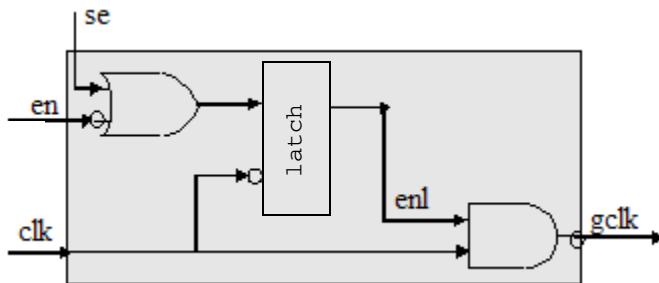
[Figure A-49](#) shows the derived `none_nedgede_control_obs_invgclk` integrated clock-gating cell circuit.

Figure A-49 none_nedgede_control_obs_invgclk Circuit**latch_posedgeactivelow_invgclk**

[Figure A-50](#) shows the derived `latch_posedgeactivelow_invgclk` integrated clock-gating cell circuit.

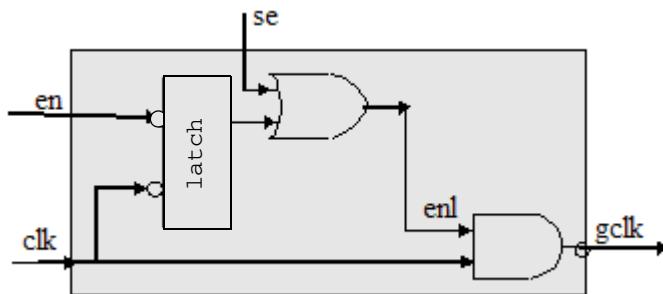
Figure A-50 latch_posedgeactivelow_invgclk Circuit**latch_posedgeactivelow_precontrol_invgclk**

[Figure A-51](#) shows the derived `latch_posedgeactivelow_precontrol_invgclk` integrated clock-gating cell circuit.

Figure A-51 *latch_posedgeactivelow_precontrol_invclk* Circuit

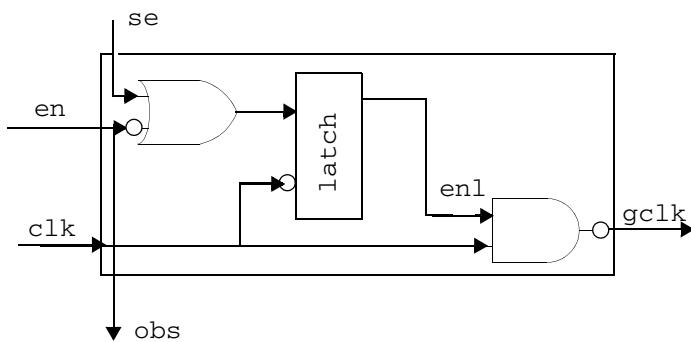
latch_posedgeactivelow_postcontrol_invclk

Figure A-52 shows the derived *latch_posedgeactivelow_postcontrol_invclk* integrated clock-gating cell circuit.

Figure A-52 *latch_posedgeactivelow_postcontrol_invclk* Circuit

latch_posedgeactivelow_precontrol_obs_invclk

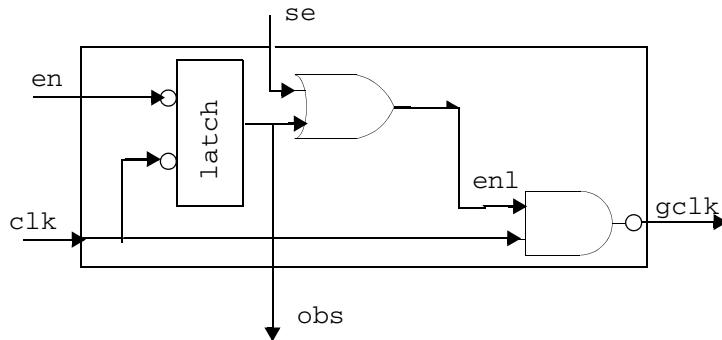
Figure A-53 shows the derived *latch_posedgeactivelow_precontrol_obs_invclk* integrated clock-gating cell circuit.

Figure A-53 *latch_posedgeactivelow_precontrol_obs_invclk* Circuit

latch_posedgeactivelow_postcontrol_obs_invclk

[Figure A-54](#) shows the derived `latch_posedgeactivelow_postcontrol_obs_invclk` integrated clock-gating cell circuit.

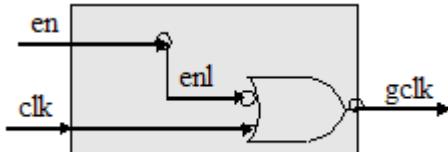
Figure A-54 latch_posedgeactivelow_postcontrol_obs_invclk Circuit



none_posedgeactivelow_invclk

[Figure A-55](#) shows the derived `none_posedgeactivelow_invclk` integrated clock-gating cell circuit.

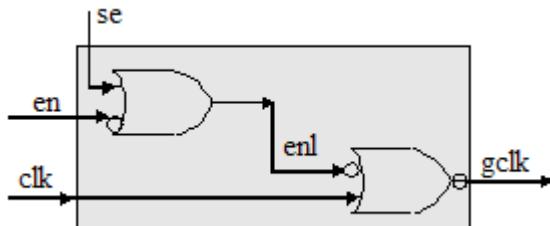
Figure A-55 none_posedgeactivelow_invclk Circuit



none_posedgeactivelow_control_invclk

[Figure A-56](#) shows the derived `none_posedgeactivelow_control_invclk` integrated clock-gating cell circuit.

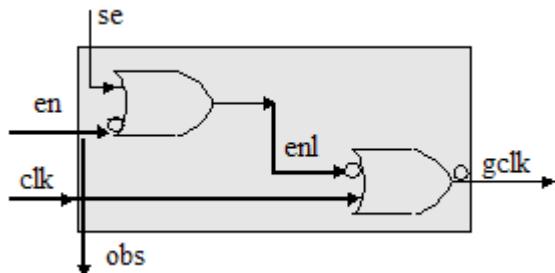
Figure A-56 none_posedgeactivelow_control_invclk Circuit



none_posedgeactive_low_control_obs_invclk

Figure A-57 shows the derived `none_posedgeactive_low_control_obs_invclk` integrated clock-gating cell circuit.

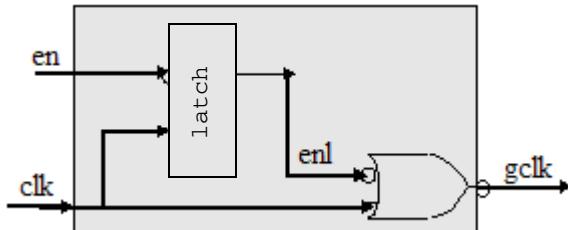
Figure A-57 *none_posedgeactive_low_control_obs_invclk Circuit*



latch_nedgedeactive_low_invclk

Figure A-58 shows the derived `latch_nedgedeactive_low_invclk` integrated clock-gating cell circuit.

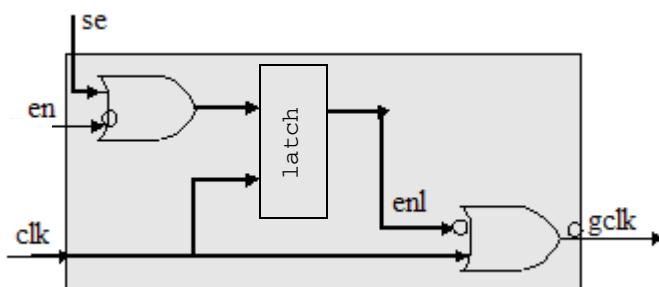
Figure A-58 *latch_nedgedeactive_low_invclk Circuit*



latch_nedgedeactive_low_precontrol_invclk

Figure A-59 shows the derived `latch_nedgedeactive_low_precontrol_invclk` integrated clock-gating cell circuit.

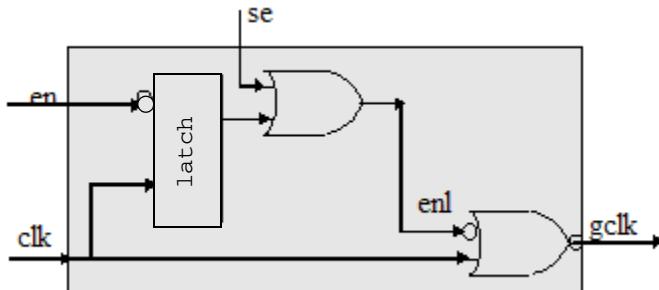
Figure A-59 *latch_nedgedeactive_low_precontrol_invclk Circuit*



latch_nedgedeactive_low_postcontrol_invgclk

Figure A-60 shows the derived `latch_nedgedeactive_low_postcontrol_invgclk` integrated clock-gating cell circuit.

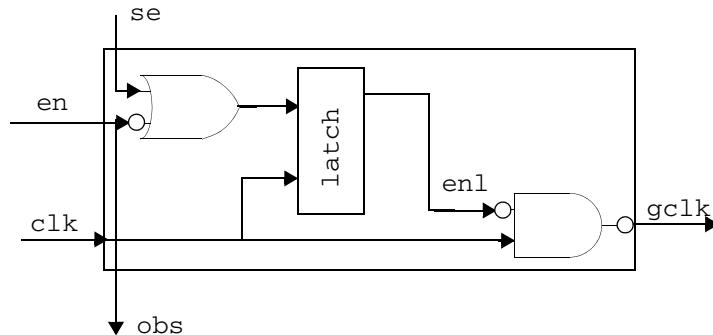
Figure A-60 *latch_nedgedeactive_low_postcontrol_invgclk Circuit*



latch_nedgedeactive_low_precontrol_obs_invgclk

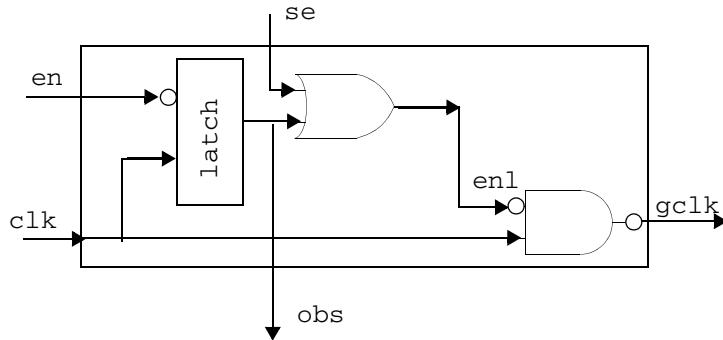
Figure A-61 shows the derived `latch_nedgedeactive_low_precontrol_obs_invgclk` integrated clock-gating cell circuit.

Figure A-61 *latch_nedgedeactive_low_precontrol_obs_invgclk Circuit*



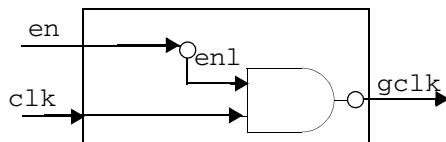
latch_nedgedeactive_low_postcontrol_obs_invgclk

Figure A-62 shows the derived `latch_nedgedeactive_low_postcontrol_obs_invgclk` integrated clock-gating cell circuit.

Figure A-62 *latch_nedgedeactivelow_postcontrol_obs_invgclk* Circuit

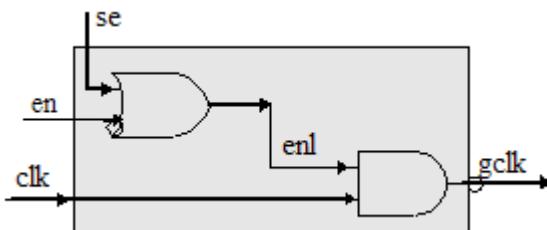
none_nedgedeactivelow_invgclk

[Figure A-63](#) shows the derived `none_nedgedeactivelow_invgclk` integrated clock-gating cell circuit.

Figure A-63 *none_nedgedeactivelow_invgclk* Circuit

none_nedgedeactivelow_control_invgclk

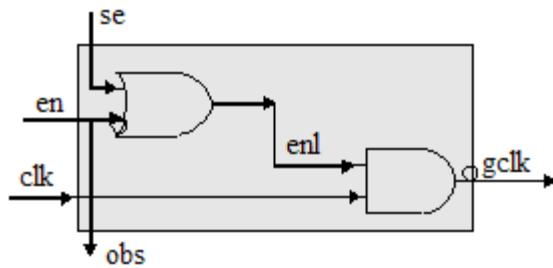
[Figure A-64](#) shows the derived `none_nedgedeactivelow_control_invgclk` integrated clock-gating cell circuit.

Figure A-64 *none_nedgedeactivelow_control_invgclk* Circuit

none_nedgedeactivelow_control_obs_invgclk

[Figure A-65](#) shows the derived `none_nedgedeactivelow_control_obs_invgclk` integrated clock-gating cell circuit.

Figure A-65 *none_nedgeactive_low_control_obs_invgclk* Circuit



B

Library Characterization Configuration

Library information is generated by characterizing the behavior of the library cells under specific conditions. You can specify these conditions in one or more of the `char_config` groups. Specifying the library characterization settings includes the following concepts and tasks explained in this chapter:

- [The `char_config` Group](#)
- [Common Characterization Attributes](#)
- [Three-State Characterization Attributes](#)
- [CCS Timing Characterization Attributes](#)
- [Input-Capacitance Characterization Attributes](#)

The char_config Group

The `char_config` group represents library characterization configuration and is a group of attributes that specify the settings to characterize a library. The library characterization settings include general and specific settings. The general settings are for common tasks, such as characterizing delays, input waveforms, output loads, and handling simulation results. The specific settings include settings for specific characterization models, such as delay, slew, constraint, power, and capacitance models.

Without the appropriate settings, library data can be misinterpreted. This can result in significant differences between the library data and SPICE simulation results. These settings are also critical for accurate recharacterization of the library.

You can define the `char_config` group within the `library`, `cell`, `pin`, and `timing` groups. You should use only one `char_config` group within each of these groups. If there are multiple `char_config` groups within a particular group, the Library Compiler tool issues an error message.

Library Characterization Configuration Syntax

[Example B-1](#) shows the general syntax for library characterization configuration.

Example B-1 General Syntax for Library Characterization Configuration

```
library (library_name) {
    char_config() {
        /* characterization configuration attributes */
    }
    ...
    cell (cell_name) {
        char_config() {
            /* characterization configuration attributes */
        }
        ...
        pin(pin_name) {
            char_config() {
                /* characterization configuration attributes */
            }
            timing() {
                char_config() {
                    /* characterization configuration attributes */
                }
            } /* end of timing */
            ...
        } /* end of pin */
        ...
    } /* end of cell */
    ...
} /* end of library */
```

The `char_config` group includes simple, and complex characterization configuration attributes.

These characterization configuration attributes are divided into the following categories:

- Common configuration
- Three-state
- Composite current source (CCS) timing
- Input capacitance

[Example B-2](#) shows the use of these attributes to document the library characterization settings.

Example B-2 Library Characterization Configuration Example

```
library (library_test) {
    lu_table_template(waveform_template) {
        variable_1 : input_net_transition;
        variable_2 : normalized_voltage;
        index_1 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7");
        index_2 ("0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
    }
    normalized_driver_waveform (waveform_template) {
        driver_waveform_name : input_driver;
        values ("0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09", \
                "0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19", \
                ...
                "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
    }
    normalized_driver_waveform (waveform_template) {
        driver_waveform_name : input_driver_cell_test;
        values ("0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09", \
                "0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19", \
                ...
                "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
    }
    normalized_driver_waveform (waveform_template) {
        driver_waveform_name : input_driver_rise;
        values ("0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09", \
                "0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19", \
                ...
                "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
    }
    normalized_driver_waveform (waveform_template) {
        driver_waveform_name : input_driver_fall;
        values ("0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09", \
                "0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19", \
                ...
                "0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9");
    }
}
```

```

char_config() {
/* library level default attributes*/
    driver_waveform(all, input_driver);
    input_stimulus_transition(all, 0.1);
    input_stimulus_interval(all, 100.0);
    unrelated_output_net_capacitance(all, 1.0);
    default_value_selection_method(all, any);
    merge_tolerance_abs( nldm, 0.1) ;
    merge_tolerance_abs( constraint, 0.1) ;
    merge_tolerance_abs( capacitance, 0.01) ;
    merge_tolerance_abs( nlpm, 0.05) ;
    merge_tolerance_rel( all, 2.0) ;
    merge_selection( all, max) ;
    three_state_disable_measurement_method : current;
    three_state_disable_current_threshold_abs : 0.05;
    three_state_disable_current_threshold_rel : 2.0;
    three_state_disable_monitor_node : tri_monitor;
    three_state_cap_add_to_load_index : true;
    ccs_timing_segment_voltage_tolerance_rel: 1.0 ;
    ccs_timing_delay_tolerance_rel: 2.0 ;
    ccs_timing_voltage_margin_tolerance_rel: 1.0 ;
    receiver_capacitance1_voltage_lower_threshold_pct_rise : 20.0;
    receiver_capacitance1_voltage_upper_threshold_pct_rise : 50.0;
    receiver_capacitance1_voltage_lower_threshold_pct_fall : 50.0;
    receiver_capacitance1_voltage_upper_threshold_pct_fall : 80.0;
    receiver_capacitance2_voltage_lower_threshold_pct_rise : 20.0;
    receiver_capacitance2_voltage_upper_threshold_pct_rise : 50.0;
    receiver_capacitance2_voltage_lower_threshold_pct_fall : 50.0;
    receiver_capacitance2_voltage_upper_threshold_pct_fall : 80.0;
    capacitance_voltage_lower_threshold_pct_rise : 20.0;
    capacitance_voltage_lower_threshold_pct_fall : 50.0;
    capacitance_voltage_upper_threshold_pct_rise : 50.0;
    capacitance_voltage_upper_threshold_pct_fall : 80.0;
    ...
}
...
cell (cell_test) {
    char_config() {
        /* input driver for cell_test specifically */
        driver_waveform (all, input_driver_cell_test);
        /* specific default arc selection method for constraint */
        default_value_selection_method (constraint, max);
        default_value_selection_method_rise(nldm_transition, min);
        default_value_selection_method_fall(nldm_transition, max);
        ...
    }
    ...
    pin(pin1) {
        char_config() {
            driver_waveform_rise(delay, input_driver_rise);
        }
    ...
    timing() {

```

```

char_config() {
    driver_waveform_rise(constraint, input_driver_rise);
    driver_waveform_fall(constraint, input_driver_fall);
    /* specific ccs segmentation tolerance for this timing arc */
    ccs_timing_segment_voltage_tolerance_rel: 2.0 ;
}
} /* timing */
} /* pin */
} /* cell */
} /* lib */

```

Common Characterization Attributes

To specify the common characterization settings, set the common configuration attributes. All common configuration attributes of the `char_config` group are complex. A complex characterization configuration attribute has the following syntax:

Syntax

`complex_attribute_name (char_model, value) ;`

The first argument of the complex attribute is the characterization model. The second argument is a value of this attribute, such as a waveform name, a specific characterization method, a numerical value of a model parameter, or other values. Use the syntax to apply the attribute value to a specific characterization model. You can specify multiple complex attributes in the `char_config` group. You can also specify a single complex attribute multiple times for different characterization models.

However, when you specify the same attribute in multiple `char_config` groups at different levels, such as at the library, cell, pin, and timing levels, the attribute specified at the lower level gets priority over the ones specified at the higher levels. For example, the pin-level `char_config` group attributes have higher priority over the library-level `char_config` group attributes. [Table B-1](#) lists the valid characterization models of the `char_config` group and their corresponding descriptions.

Table B-1 Valid Characterization Models of the `char_config` Group

Characterization model	Description
<code>all</code>	Default model. The <code>all</code> model has the lowest priority among the characterization models. Any other characterization model overrides the <code>all</code> model.
<code>ndlml</code>	Nonlinear delay model (NLDM)
<code>nldm_delay</code>	Specific NLDMs with higher priority over the default NLDM
<code>nldm_transition</code>	

Table B-1 Valid Characterization Models of the char_config Group (Continued)

Characterization model	Description
capacitance	Capacitance model
constraint	Constraint model
constraint_setup	Specific constraint models with higher priority over the general constraint model
constraint_hold	
constraint_recovery	
constraint_removal	
constraint_skew	
constraint_min_pulse_width	
constraint_no_change	
constraint_non_seq_setup	
constraint_non_seq_hold	
constraint_minimum_period	
nlpm	Nonlinear power model (NLPM)
nlpm_leakage	Specific NLPMs with higher priority over the general NLPM
nlpm_input	
nlpm_output	

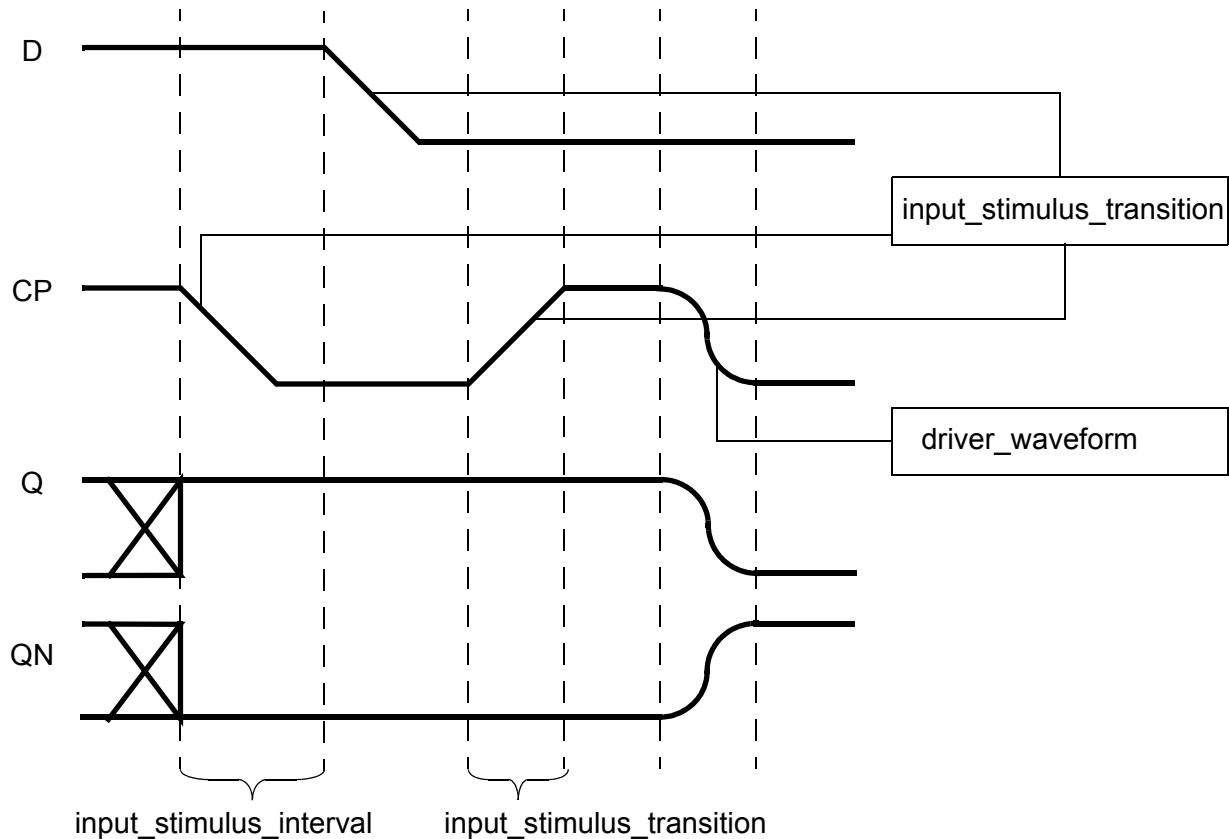
Example B-3 shows the syntax of the characterization configuration complex attributes.

Example B-3 Syntax of Common Configuration Complex Attributes in the char_config Group

```
char_config() {
    driver_waveform(char_model, waveform_name);
    driver_waveform_rise(char_model, waveform_name);
    driver_waveform_fall(char_model, waveform_name);
    input_stimulus_transition(char_model, float);
    input_stimulus_interval(char_model, float);
    unrelated_output_net_capacitance(char_model, float);
    default_value_selection_method(char_model, method);
    default_value_selection_method_rise(char_model, method);
    default_value_selection_method_fall(char_model, method);
    merge_tolerance_abs(char_model, float);
    merge_tolerance_rel(char_model, float);
    merge_selection(char_model, method);
    ...
}
```

Figure B-1 illustrates the use of `driver_waveform`, `input_stimulus_transition`, and `input_stimulus_interval` attributes for the delay arc characterization of a D flip-flop.

Figure B-1 Delay Arc Characterization



In [Figure B-1](#), an input stimulus with multiple transitions characterizes the delay arc, CP to Q or QN. The `input_stimulus_transition` and `input_stimulus_interval` attributes define and control the transitions and corresponding intervals, respectively. The `driver_waveform` attribute controls the last transition of the clock pulse, CP.

driver_waveform Attribute

The `driver_waveform` attribute defines the driver waveform to characterize a specific characterization model.

You can define the `driver_waveform` attribute within the `char_config` group at the library, cell, pin, and timing levels. If you define the `driver_waveform` attribute within the `char_config` group at the library level, the library-level `normalized_driver_waveform` group is ignored when the `driver_waveform_name` attribute is not defined.

If you define the `driver_waveform` attribute both within and out of the `char_config` group, the Library Compiler tool issues a warning message.

If you do not define this attribute in the `char_config` group, the ramp waveform is used by default.

driver_waveform_rise and driver_waveform_fall Attributes

The `driver_waveform_rise` and `driver_waveform_fall` attributes define a specific rising and falling driver waveform, respectively, for a specific characterization model.

You can define the `driver_waveform_rise` and `driver_waveform_fall` attributes within the `char_config` group at the library, cell, pin, and timing levels. If you define the `driver_waveform_rise` and `driver_waveform_fall` attributes within the `char_config` group at the library level, the library-level `normalized_driver_waveform` group is ignored when the `driver_waveform_name` attribute is not defined.

If you use the `driver_waveform_rise` or `driver_waveform_fall` attributes both within and out of the `char_config` group, the Library Compiler tool issues a warning message.

If you do not define these attributes in the `char_config` group, the ramp waveform is used by default.

input_stimulus_transition Attribute

The `input_stimulus_transition` attribute specifies the transition time for all the input-signal edges except the arc input pin's last transition, during generation of the input stimulus for simulation. For example, in [Figure B-1 on page B-7](#), the last transition of the clock pulse, CP, uses the `driver_waveform` attribute, and not the `input_stimulus_transition` attribute.

The time units of the `input_stimulus_transition` attribute are specified by the library-level `time_unit` attribute.

You must define this attribute. Otherwise, the Library Compiler tool issues an error message.

input_stimulus_interval Attribute

The `input_stimulus_interval` attribute specifies the time-interval between the input-signal toggles to generate the input stimulus for a characterization cell. The time units of this attribute are specified by the library-level `time_unit` attribute.

You must define the `input_stimulus_interval` attribute. Otherwise, the Library Compiler tool issues an error message.

unrelated_output_net_capacitance Attribute

The `unrelated_output_net_capacitance` attribute specifies a load value for an output pin that is not a related output pin of the characterization model. The valid value is a floating-point number, and is defined by the library-level `capacitive_load_unit` attribute.

If you do not specify this attribute for the `nldm_delay` and `nlpm_output` characterization models, the unrelated output pins use the load value of the related output pin. However, you must specify this attribute for any other characterization model. Otherwise, the Library Compiler tool issues an error message.

default_value_selection_method Attribute

The `default_value_selection_method` attribute defines the method of selecting a default for

- The delay arc from state-dependent delay arcs.
- The constraint arc from state-dependent constraint arcs.
- Pin-based minimum pulse-width constraints from simulated results with side pin combinations.
- Internal power arcs from multiple state-dependent `internal_power` groups.
- The `cell_leakage_power` attribute from the state-dependent values in leakage power models.
- The input-pin capacitance from capacitance values for input-slew values used for timing characterization.

default_value_selection_method_rise and default_value_selection_method_fall Attributes

Use the `default_value_selection_method_rise` and `default_value_selection_method_fall` attributes when the selection methods for rise and fall are different.

You must define either the `default_value_selection_method` attribute, or the `default_value_selection_method_rise` and `default_value_selection_method_fall` attributes. Otherwise, the Library Compiler tool issues an error message. [Table B-2](#) lists the valid selection methods for the

`default_value_selection_method`, `default_value_selection_method_rise`, `default_value_selection_method_fall`, and `merge_selection` attributes and their descriptions.

Table B-2 Valid Common Configuration Selection Methods

Method	Description
<code>any</code>	Selects a random value from the state-dependent data
<code>min</code>	Selects the minimum value from the state-dependent data at each index point.
<code>max</code>	Selects the maximum value from the state-dependent data at each index point.
<code>average</code>	Selects an average value from the state-dependent data at each index point.
<code>min_mid_table</code>	When the state-dependent data is a lookup table (LUT), this method selects the minimum value from the LUT. The minimum value is selected by comparing the middle value in the LUT, with each of the table-values. Note: The middle value corresponds to an index value. If the number of index values is odd, then the middle value is taken as the median value. However, if the number of index values is even, then the smaller of the two values is selected as the middle value.
<code>max_mid_table</code>	When the state-dependent data is a lookup table (LUT), this method selects the maximum value from the LUT. The maximum value is selected by comparing the middle value in the LUT, with each of the table-values. Note: The middle value corresponds to an index value. If the number of index values is odd, then the middle value is taken as the median value. However, if the number of index values is even, then the smaller of the two values is selected as the middle value.
<code>follow_delay</code>	Selects the value from the state-dependent data for delay selection. This method is valid only for the <code>nldm_transition</code> characterization model, that is, the <code>follow_delay</code> method applies specifically to default transition-table selection and not any other default selection.

merge_tolerance_abs and merge_tolerance_rel Attributes

The `merge_tolerance_abs` and `merge_tolerance_rel` attributes specify the absolute and relative tolerances, respectively, to merge arc simulation results. Specify the absolute tolerance value in the corresponding library unit, and the relative tolerance value in percent, for example, 10.0 for 10 percent.

If you specify both the `merge_tolerance_abs` and `merge_tolerance_rel` attributes, the results are merged if either or both the tolerance conditions are satisfied. If you do not specify any of these attributes, data including identical data is not merged.

merge_selection Attribute

The `merge_selection` attribute specifies the method of selecting the merged data. When multiple sets of state-dependent data are merged, the attribute selects a particular set of the state-dependent data to represent the merged data. See [Table B-2 on page B-10](#) for the valid methods and their descriptions of the `merge_selection` attribute.

You must define the `merge_selection` attribute if you have defined either of the `merge_tolerance_abs` or `merge_tolerance_rel` attributes. Otherwise, the Library Compiler tool issues an error message.

Three-State Characterization Attributes

To specify the three-state characterization settings, set the simple attributes that define three-state characterization. [Example B-4](#) shows the syntax of these simple attributes.

Example B-4 Syntax for Three-State Simple Attributes in the char_config Group

```
char_config() {
    three_state_disable_measurement_method : voltage | current;
    three_state_disable_current_threshold_abs : value;
    three_state_disable_current_threshold_rel : value;
    three_state_disable_monitor_node : string;
    three_state_cap_add_to_load_index : true | false;
    ...
}
```

three_state_disable_measurement_method Attribute

The `three_state_disable_measurement_method` attribute defines the method to identify the three-state condition of a pin.

Use either of the following methods to identify the three-state condition on the pin.

- `voltage`: measures the voltage waveform to the gate input of the three-state stage
- `current`: measures the leakage current flowing through the pullup and pulldown resistors of the three-state stage

Therefore, the valid values of this attribute are `voltage` and `current`.

In a `pin` group, this attribute is valid only for a three-state pin. You must define this attribute if the library contains one or more three-state cells. Otherwise, the Library Compiler tool issues an error message.

`three_state_disable_current_threshold_abs` and `three_state_disable_current_threshold_rel` Attributes

The `three_state_disable_current_threshold_abs` and `three_state_disable_current_threshold_rel` attributes, respectively, specify the absolute and relative current threshold values to distinguish between the low- and high-impedance states of a three-state output pin.

The unit of the absolute current threshold value is specified in the `current_unit` attribute of the `library` group. The relative current threshold value is specified as a percentage of the peak current, for example, 100.0 for 100 percent of the peak current.

In the `pin` group, these attributes are valid only for an inout pin. If you specify both the `three_state_disable_current_threshold_abs` and `three_state_disable_current_threshold_rel` attributes, the three-state is identified upon reaching either of the two threshold values.

You must specify at least one of the `three_state_disable_current_threshold_abs` and `three_state_disable_current_threshold_rel` attributes for the `current` method. Otherwise, the Library Compiler tool issues an error message.

`three_state_disable_monitor_node` Attribute

The `three_state_disable_monitor_node` attribute specifies the internal node name to probe for the three-state voltage measurement method.

In the `pin` group, this attribute is valid only for an inout pin. You must define this attribute for the `voltage` method. Otherwise, the Library Compiler tool issues an error message.

`three_state_cap_add_to_load_index` Attribute

The `three_state_cap_add_to_load_index` attribute indicates that the pin-capacitance of a three-state pin is added to each index value of the `total_output_net_capacitance` variable.

You must define this attribute. Otherwise, the Library Compiler tool issues an error message.

CCS Timing Characterization Attributes

To specify the CCS timing characterization settings, set the simple attributes that define CCS timing generation. [Example B-5](#) shows the syntax of these simple attributes.

Example B-5 Syntax of CCS Timing Simple Attributes

```
char_config() {
    ccs_timing_segment_voltage_tolerance_rel: float;
    ccs_timing_delay_tolerance_rel: float;
    ccs_timing_voltage_margin_tolerance_rel: float;
    receiver_capacitance1_voltage_lower_threshold_pct_rise : float;
    receiver_capacitance1_voltage_upper_threshold_pct_rise : float;
    receiver_capacitance1_voltage_lower_threshold_pct_fall : float;
    receiver_capacitance1_voltage_upper_threshold_pct_fall : float;
    receiver_capacitance2_voltage_lower_threshold_pct_rise : float;
    receiver_capacitance2_voltage_upper_threshold_pct_rise : float;
    receiver_capacitance2_voltage_lower_threshold_pct_fall : float;
    receiver_capacitance2_voltage_upper_threshold_pct_fall : float;
    ...
}
```

You must define all these attributes if the library includes a CCS model. Otherwise, the Library Compiler tool issues an error message.

ccs_timing_segment_voltage_tolerance_rel Attribute

The `ccs_timing_segment_voltage_tolerance_rel` attribute specifies the maximum permissible voltage difference between the simulation waveform and the CCS waveform to select the CCS model point. The floating-point value is specified in percent, where 100.0 represents 100 percent maximum permissible voltage difference.

ccs_timing_delay_tolerance_rel Attribute

The `ccs_timing_delay_tolerance_rel` attribute specifies the acceptable difference between the CCS waveform delay and the delay measured from simulation. The floating-point value is specified in percent, where 100.0 represents 100 percent acceptable difference.

ccs_timing_voltage_margin_tolerance_rel Attribute

The `ccs_timing_voltage_margin_tolerance_rel` attribute specifies the voltage tolerance to determine whether a signal has reached the rail-voltage value. The floating-point value is specified as a percentage of the rail voltage, such as 96.0 for 96 percent of the rail voltage.

CCS Receiver Capacitance Attributes

The following attributes specify the current integration limits, as a percentage of the voltage, to calculate the CCS receiver capacitances:

- `receiver_capacitance1_voltage_lower_threshold_pct_rise`
- `receiver_capacitance1_voltage_upper_threshold_pct_rise`
- `receiver_capacitance1_voltage_lower_threshold_pct_fall`
- `receiver_capacitance1_voltage_upper_threshold_pct_fall`
- `receiver_capacitance2_voltage_lower_threshold_pct_rise`
- `receiver_capacitance2_voltage_upper_threshold_pct_rise`
- `receiver_capacitance2_voltage_lower_threshold_pct_fall`
- `receiver_capacitance2_voltage_upper_threshold_pct_fall`

The floating-point values of these attributes can vary from 0.0 to 100.0.

Input-Capacitance Characterization Attributes

To specify input-capacitance characterization settings, set the simple attributes that define input-capacitance measurement. [Example B-6](#) shows the syntax of these simple attributes.

Example B-6 Syntax of Input-Capacitance Characterization Simple Attributes

```
char_config() {
    capacitance_voltage_lower_threshold_pct_rise : float;
    capacitance_voltage_lower_threshold_pct_fall : float;
    capacitance_voltage_upper_threshold_pct_rise : float;
    capacitance_voltage_upper_threshold_pct_fall : float;
    ...
}
```

Each floating-point threshold value is specified as a percentage of the supply voltage, and can vary from 0.0 to 100.0.

You must define all the simple attributes mentioned in [Example B-6](#), in the `char_config` group for input-capacitance characterization. Otherwise, the Library Compiler tool issues an error message.

`capacitance_voltage_lower_threshold_pct_rise` and `capacitance_voltage_lower_threshold_pct_fall` Attributes

The `capacitance_voltage_lower_threshold_pct_rise` and `capacitance_voltage_lower_threshold_pct_fall` attributes specify the lower-threshold value of a rising and falling voltage waveform, respectively, for calculating the NLDM input-pin capacitance.

`capacitance_voltage_upper_threshold_pct_rise` and `capacitance_voltage_upper_threshold_pct_fall` Attributes

The `capacitance_voltage_upper_threshold_pct_rise` and `capacitance_voltage_upper_threshold_pct_fall` attributes specify the upper-threshold value of a rising and falling voltage waveform, respectively, for calculating the NLDM input-pin capacitance.

